NASA Technical Memorandum 4662

Transferring Ecosystem Simulation Codes to Supercomputers

J. W. Skiles and C. H. Schulbach

February 1995

National Aeronautics and
Space Administration

NASA Technical Memorandum 4662

Transferring Ecosystem Simulation Codes to Supercomputers

J. W. Skiles, *Johnson Controls World Services, Inc., Cape Canaveral, Florida*
C. H. Schulbach, *Ames Research Center, Moffett Field, California*

February 1995

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

CONTENTS

Transferring Ecosystem Simulation Codes to Supercomputers

J. W. SKILES[*] AND C. H. SCHULBACH

Ames Research Center

Summary

Many computer codes have been developed for the simulation of ecological systems in the last twenty-five years. This development took place initially on main-frame computers, then mini-computers, and more recently, on micro-computers and workstations. Recent recognition of earth system science as a High Performance Computing and Communications Program Grand Challenge area emphasizes supercomputers (both parallel and distributed systems) as the next set of tools for ecological simulation. Transferring ecosystem simulation codes to such systems is not a matter of simply compiling and executing existing code on the supercomputer, since significant differences exist between the system architectures of sequential, scalar computers and parallel and/or vector supercomputers. To more effectively match the application to the architecture and achieve reasonable performance, the parallelism, if it exists, of the original application must be exploited. We discuss our work in transferring a general grassland simulation model (developed on a VAX in the FORTRAN computer programming language) to a Cray Y-MP/C-90. We show the Cray shared-memory vector architecture and discuss our rationale for selecting the Cray. We describe porting the model to the Cray and executing and verifying a baseline version, and we discuss the changes we made to exploit the parallelism in the application and to improve code execution. As a result of these efforts, the Cray executed the model 30 times faster than the VAX 11/785 and 10 times faster than a Sun 4 workstation. We achieved an additional speed increase of approximately 30 percent over the original Cray run by using the compiler's vectorizing capabilities and the machine's ability to put subroutines and functions "in-line" in the code. With the modifications, the code still runs at about five percent of the Cray's peak speed because it makes ineffective use of the vector and parallel processing capabilities of the Cray. By restructuring the code to increase vectorization and parallelization, we believe we could execute the code six to ten times faster than the current Cray version.

[*]Johnson Controls World Services, Inc., Cape Canaveral, Florida.

Introduction

Scientists involved in ecosystem studies have used models for many years. Models help explain processes in ecosystems that cannot be observed or measured explicitly. They serve to direct study to areas where data and understanding are missing. Further, models allow manipulation of simulated ecosystems not feasible with the actual system because of time, budget, or conservation constraints. The tools for ecosystem modeling, especially computing platforms, were also evolving simultaneously.

Ecosystem model development began with the use of mainframe computational platforms (ref. 1). Models were submitted to a queue in the form of card decks, executed in batch, and output returned as hardcopy, hours or days later. Mini-computers eased the turnaround time between job executions because they were more affordable. Since there were more of them, they usually operated in a time-sharing mode, and they offered greater access to graphic and peripheral plotting devices.

The advent of the micro-processor brought computing to the individual user. Large facilities were no longer necessary for computing, and many specialized output devices and applications became available for the display of model output. Ecosystem modelers, over this same period of time, have continued to demand faster and faster rates of execution and more and more core or random access memory (RAM). Supercomputing platforms would seem to meet these continually rising demands.

Though signs of documentation of supercomputer use are beginning to appear in the ecological literature (refs. 2–4), supercomputers, generally located in large facilities, were bypassed and never fully embraced by the ecosystem modeling community. (We except here those ecosystem models that use output from or are linked to global climate models (GCMs), since many use supercomputers for execution (refs. 5 and 6).

Bypassing of supercomputers by the ecosystem modeling community occurred for a variety of reasons: supercomputer time is perceived as expensive and difficult to obtain because of the paperwork needed to open and maintain an account; supercomputers are a limiting resource for modelers and CPU time is not always available at accessible installations; control languages and compilers are often different from standards in the computer industry and necessitate the user learning new commands in order to execute a model. In addition, network connections used to transfer code, data, and output between the supercomputer and the user's front-end computer (perhaps a workstation or desktop computer) are slow, difficult to use, and again require the user to learn new commands. Finally, the supposed decrease in execution time of model codes is not fully realized because of the above considerations and because the ported code often does not take advantage of supercomputer architecture.

A growing emphasis on the grand challenges in ecological modeling is changing the use of supercomputers in the field. Ecosystem science is included in the Earth and Space Sciences grand challenge area for NASA in the High Performance Computing and Communications Program (ref. 7). One such grand challenge is determining the global carbon balance. Models designed for calculating this balance (ref. 8, for example) use large data sets for initializing and driving the simulation. Large data matrices holding intermediate and state variables are also maintained and manipulated during these simulations, necessitating large amounts of RAM and fast execution times in order to perform large numbers of simulation experiments.

Figure 1 shows the development of computer capabilities over the last two decades and the projected Earth and Space Sciences Computing Requirements in terms of speed and memory. NASA now emphasizes the use of supercomputers (both parallel and distributed systems) as the next tool for ecological simulation and is making supercomputer platforms more readily available to ecosystem modelers. Ecosystem modelers would benefit from the use of supercomputers because they could more readily simulate large geographical areas with reasonable turnaround time; be able to execute large unit time simulations (many days or years) ordinarily taking too much time on a lower-level computing platform; be able to use large remote sensing data sets to drive the model or use as validation; and be able to do more model scenario testing or gaming with fewer real or clock-time constraints.

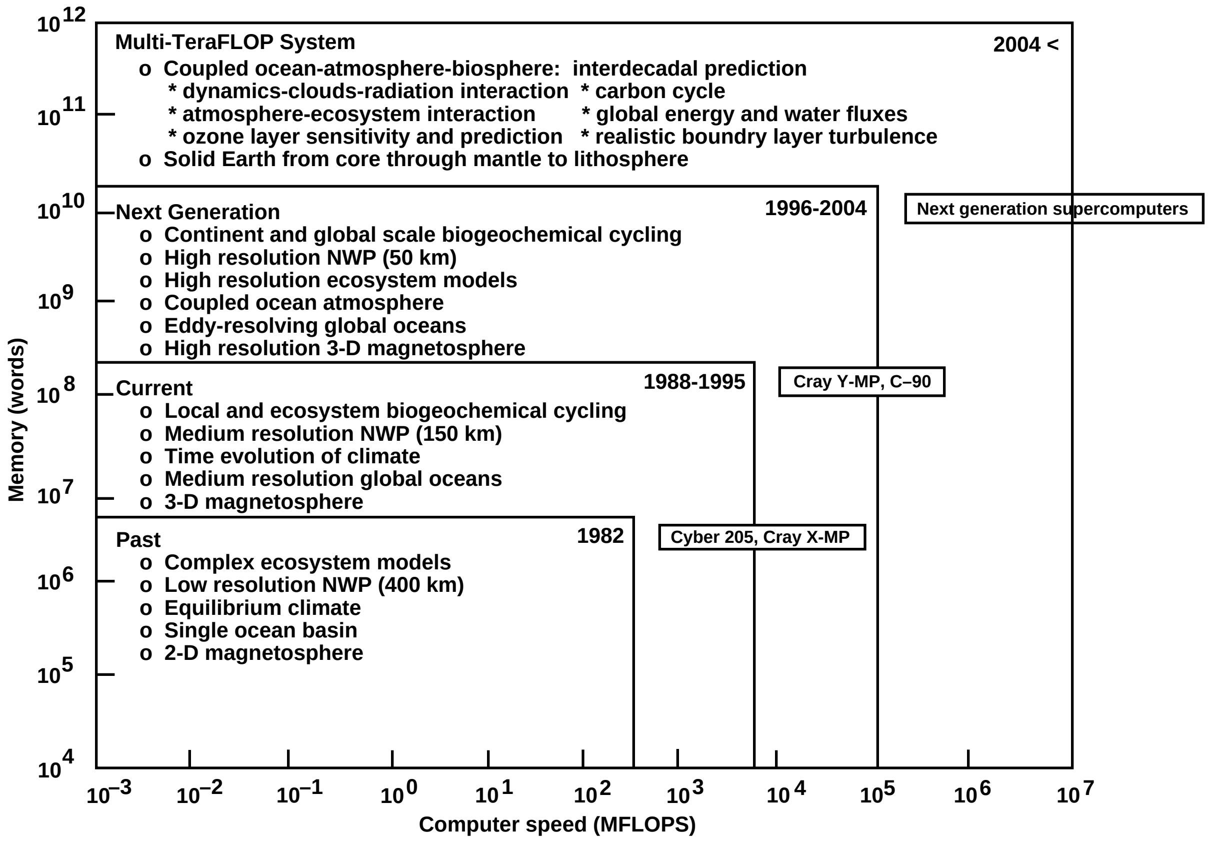

Figure 1. Past, present, and predicted earth and space sciences computing requirements displayed as computer speed versus words of memory. MFLOPS is millions of floating point operations per second; NWP is numerical weather prediction. (Redrawn from J. Bredekamp and J. Harris, NASA Headquarters; personal communication.)

The need for the advanced computational systems shown in figure 1 may not be apparent now, but the projected increase in computing capabilities will enable ecologists to attack problems that are thought to be too large or too difficult today (ref. 9). For example, statistical mechanical theories of interacting species have been developed in community and ecosystem ecology. However, the theories have had little impact on the discipline because the necessary simplifying assumptions cannot be justified biologically (ref. 10). The simplifications have been imposed by computational limitations. The further development of predictive models in the earth and biological sciences will continue to strain the capacity of the most powerful computers. These models will require new techniques for handling wide spatial and temporal scales, stiff systems of equations, the processing of very large volumes of data, and advanced distributed data management and information systems (ref. 11), as well as techniques for the visual presentation of model results.

In this paper, we address the concern of maximizing use of supercomputer architectures. We use an ecosystem simulation model (see Ecosystem Model Description section) constructed on a Digital Equipment Corporation (DEC) VAX[1] mini-computer and tested on several mainframe computers to standardize the code and to establish conformity of output. We describe transferring that computer code to a Cray C90 supercomputer. We detail the obstacles we encountered in this transfer, our solutions, and the changes in code structure we made in order to maximize processor use and minimize CPU time.

Assumptions

Before we began this work, we made some assumptions pertaining to porting existing models to supercomputers. The first assumption is that the model performs as it was designed to perform on lower-level computing platforms. It makes no sense to move a model to a supercomputer if it is not functioning properly on the original machine. The model we used met this criterion.

The next assumption is that the users who are moving the model have knowledge of the model and of the data required to execute the model. The users need to know about the model in order to detect errors in execution or output once the model is running on the supercomputer. The users also need to know how many and what type of files are required to initialize and execute the model. This familiarity helps in understanding the input/output (I/O) data structure(s) utilized and produced by the model. Users should have complete output from the model that was produced on a low-level platform for comparison before the model is moved to a supercomputer.

The users are also expected to have good documentation in hand in order to track down output deviations from the standard output once the model is ported. Too often model documentation consists of notes and scribbles collected in a loose-leaf binder, or it resides in the modeler's head and is not available to other users. Good documentation is especially important if the users doing the porting to a supercomputer do not have first-hand experience with the model.

Lastly, we assumed that we would have access to a supercomputing platform. As mentioned above, supercomputing resources are often limited, and we needed to be able to access the chosen machine over a local or wide area network as required.

Earlier drafts of this manuscript benefited from contributions by Hector D'Antoni, Christopher Potter, and Don Sullivan.

Computer Platform Description

When computers were first introduced, they generally followed what is now referred to as the "von Neumann" architecture, after John von Neumann who proposed this computer architecture in the 1940s (ref. 12). This architecture consists of a control unit (program counter and instruction fetcher), the processor (arithmetic and logic unit), and a memory containing the program and data. Operations are performed one at a time in the order they are encountered. Since this method is inherently limited, it was not long before a number of concepts were proposed to enhance the speed of execution. The result is today's high performance computers that can process data at speeds exceeding GFLOPS (billions of floating point operations per second). These speeds are obtained through replication of processors and the use of other techniques (see next section). However, such GFLOPS speeds cannot be obtained unless applications can make use of the parallelism of the architecture. In fact, many applications may achieve only a few percent of the top speed of the supercomputing machine.

We next explain some of the methods for exploiting parallelism in computer architecture, provide our rationale for choosing the Cray, and give a short overview of the architecture of the Cray C90. More detailed information on computer architecture and the Cray systems can be obtained from references 13–15.

[1]Use of trade names in this paper is for convenience only and does not imply endorsement by the National Aeronautics and Space Administration or by the U.S. Government.

Parallelism in Computer Architecture

The use of separate processors, introduced to handle input and output (I/O) functions, was one of the first examples of parallelism in computer architecture. Later, interleaved or banked memories were used to improve data access speed, and independent functional units were created in which machine functions were assigned to specialized units able to execute simultaneously. The CDC 7600, for example, had independent functional units: floating-point add, long add, floating-point multiply, floating-point divide, increment, shift, Boolean, normalize, and population count. These functional units could be pipelined as well.

Pipelining consists of using assembly line techniques to increase throughput. Tasks are divided into stages, and separate pairs of operands occupy different stages simultaneously. Figure 2 shows a possible pipeline for floating-point addition. If each stage of the pipeline requires 10 nanoseconds, then, at the beginning, the first pair of operands enters the pipe; the next pair enters 10 nanoseconds later, and so on. After 40 nanoseconds, the first result is produced. Every 10 nanoseconds after that, another result is produced. Pipeline techniques can apply not only to the execution of instructions, but to instruction processing as well. Instruction processing can be divided into phases such as instruction fetch, instruction decode, and operand fetch.

Computer systems available in the mid- to late-1960s used many of the techniques mentioned here to improve processing speed. During this period, computer architects realized that parallelism could be achieved in additional ways. Flynn (ref. 16) proposed a classification scheme that related machines and their instructions.

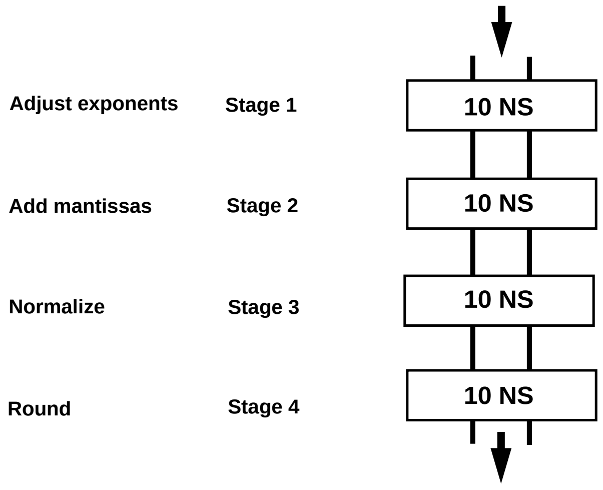

Figure 2. Example of a pipeline for floating-point addition. The operation begins at the upper right.

Flynn's Classification Scheme

Figure 3 summarizes Flynn's classification scheme. This scheme describes the function of a computer system, not its architecture, but it is widely used to provide a framework for discussion. A stream is a sequence of items, either instructions or data, operated on by a processor. There can be a single stream of instructions or data, or multiple streams. The von Neumann architecture is in the single instruction stream, single data stream (SISD) machine class. All non-von Neumann machines fall into either the single instruction stream, multiple data stream (SIMD) or multiple instruction stream, multiple data stream (MIMD) class because the MISD class is generally considered empty.

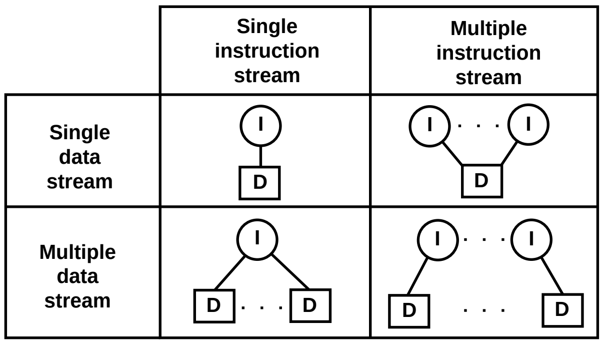

Figure 3. Flynn's taxonomy of computer architecture.

Within the SIMD class, two general types of architecture occur. The vector, or pipeline, processor represents an extension of the idea of pipelined functional units. There is still only one control unit issuing instructions, but one instruction can cause an operation to be carried out on a sequence of elements. The operations are done in a pipelined manner. The Cray 1 and Cyber 205 computers are examples of pipelined or vector processors.

The other type of SIMD machine is the array, or parallel, processor. It is characterized by replicated processing elements directly connected to a single common, control unit. Each processing element has its own registers and storage. The processors operate in lockstep under control of the single control unit. Early examples of array processors are the ILLIAC IV and the Massively Parallel Processor; a more recent example is the Thinking Machines Connection Machine CM-2.

The MIMD machines, or multiprocessors, consist of multiple processors, each obeying its own instructions. As with SIMD machines, there are two general approaches to this class: (1) shared memory multiprocessors, and (2) distributed memory multiprocessors.

In shared memory multiprocessors, a processing element has a control unit and an arithmetic unit, and the processing elements share a common memory. There may be a multi-ported connection to the memory element(s), or the processing elements may be connected through some sort of connection network or switching network. The Cray X-MP, Cray Y-MP, and Cray 2 computers are examples of this kind of architecture.

With distributed memory multiprocessors, each processor has its own control unit, memory, and arithmetic unit. These processors may be connected in a variety of fashions not discussed here. Data is local to a processor, and communication is via explicit message passing. Examples of such machines include the Intel iPSC/860, Delta, and Paragon, and the Thinking Machines Connection Machine CM-5.

Selecting the Cray Platform

We selected the Cray C90 for our initial effort in porting and modifying our ecosystem model based on four considerations: (1) potential vectorization capability and parallelism with multiple processors, (2) availability, (3) ease in porting, and (4) maturity of software. Although we had other platforms available (such as the Intel iPSC/860 and Thinking Machines CM-2), we decided that the Cray would be best for establishing an initial baseline output and execution time for the model. Using the Cray would allow us to invest less time learning the computer system and more time executing and optimizing the model. The characteristics of Cray systems are well known, and the software is very mature. Also, many experienced supercomputer users acknowledge that getting their codes to run well on a Cray is the first step in getting them to run well on other supercomputers.

Cray Architecture

The Cray 1, the first commercially successful vector processor, was delivered to computer users in 1976. It included multiple, special purpose, pipelined functional units that could operate concurrently. The Cray 1 had 8 vector registers, each with 64 64-bit words, a radical departure from the 16-bit and 32-bit sequential machines in use at the time. Along with the vector registers were additional machine instructions for manipulating the vectors as units. Operations took place from one register to another. The registers received data from and sent data to main memory using starting location and an increment (≥ 1). The main memory for the original machines consisted of 1 million words divided into 16 banks that could operate concurrently. The section on data representation explains more about the characteristics of Cray floating-point arithmetic.

The Cray 1 had a special feature called “chaining” that helped increase the speed of computation. Chaining provided the ability to link vector operations so that they operated as one continuous pipeline. The result of a vector instruction was fed directly into the pipeline for the next instruction without waiting for the first instruction to complete arithmetic on all elements. Thus, once the chained pipeline was filled, multiple operations were completed each clock cycle.

The vector operations and the chaining capability resulted in a peak performance for the Cray 1 of 160 million floating-point operations per second (MFLOPS). The rate of 160 MFLOPS assumes that both the multiply and add functional units could produce a result each clock cycle (12.5 nanoseconds).

The Cray C90, the newest of the Cray products, can contain as many as 16 processors and up to 1024 million words of shared memory (approximately 8 billion bytes). Each processor of the C90 is a vector processor similar to the original Cray 1. Data representation and binary floating-point arithmetic differ very slightly from the Cray 1. However, there are notable changes in the Cray C90. It has dual (instead of single) vector pipelines (dual sets of functional units—add, multiply, reciprocal approximation) per CPU. Vector registers contain 128 (instead of 64) elements (64-bit words). In addition, the clock speed is reduced to approximately 4.2 nanoseconds.

The differences in vector length between the Cray 1 and the C90 mean that the C90 achieves its peak speed on vectors of length 128 (or multiples thereof) rather than on vectors of length 64. Another difference is that twice as many results can be produced per clock cycle (i.e., 2 adds and 2 multiplies rather than 1 add and 1 multiply for the Cray 1). (Reciprocal approximations are not counted here.) With the reduced clock cycle time of 4 nanoseconds, a single C90 processor has a peak speed of approximately 1 GFLOPS. Combining all 16 processors results in a capability of over 16 GFLOPS.

Ecosystem Model Description

We used the Simulation of Production and Utilization of Rangelands (SPUR) model (ref. 17) in our work. It is a rangeland ecosystem model composed of modules simulating rangeland hydrology, snow accumulation and melt, plant growth and mortality, and herbivore/plant/soil interactions. The SPUR model has a dynamic hydrology module (ref. 18) and a plant growth module (ref. 19). Each module is based on physical processes known to occur in rangeland ecosystems. SPUR generally operates on a daily time step, even though some processes are simulated on shorter time spans and integrated over the entire day. The model is driven by daily maximum and minimum temperatures, daily precipitation, daily solar radiation, and a daily wind value. In the absence of an actual weather record, the variables can be provided by a stochastic weather generator (ref. 20). The hydrology module supplies the plant module with soil moisture tension by soil layer, and the plant component supplies the hydrology component with leaf area index (LAI) (fig. 4). The plant component explicitly models carbon and nitrogen flux from the atmosphere through standing green vegetation, live roots, dead roots, propagules, standing dead vegetation, soil organic matter, and litter. Nitrogen accounting also considers mineralization and soil inorganic concentrations. In addition, the model includes modules for domestic and wild herbivore grazing and for rangeland economics (fig. 4), though we did not use these modules in this exercise.

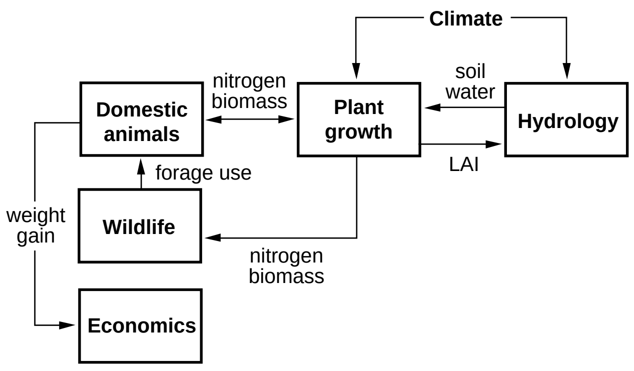

Figure 4. Major modules for the SPUR model.

Model Validation

The SPUR model and model components have been subjected to a number of validation tests. Renard (ref. 21) and Springer (ref. 18) tested the hydrology module and reported that SPUR can adequately reproduce seasonal runoff in arid watersheds. Cooley (ref. 22) evaluated the snow dynamics in the model and found good agreement between observed and predicted snow accumulation and snow melt over three seasons. Skiles (ref. 23) simulated the growth of the two dominant grasses in the shortgrass steppe of Colorado and concluded that the plant module adequately reproduced the biomass production of the grasses and matched the dynamics of the growing season. Hanson (ref. 19) showed that the plant-animal interface in the SPUR model correctly predicted domestic animal weight gains as a function of stocking rate for a Colorado grassland.

SPUR has been successfully used to predict plant biomass production on pastures in West Virginia (ref. 24), provide simulated forage for a modified domestic animal module (ref. 25), and supply biomass to a grazing behavior model (ref. 26). A geographic information system (GIS) has been used with the SPUR plant and hydrology components to demonstrate that high orders of stream complexity are not necessary to adequately simulate monthly stream runoff in Arizona (ref. 27). The SPUR model has also been used to estimate the effects of climate change on plant and livestock production in the Great Plains of North America (ref. 28) and estimate the effects of climate change and CO_2 increase on small-watershed hydrology (ref. 29).

Ecosystem Model Structure

The model consists of 3,200 lines of FORTRAN code in 43 modules (program, subroutines, and function subprograms). Approximately one-third of the code consists of non-executable records consisting of comment records and common blocks.

The SPUR model was released in two versions, a field-scale version and a basin-scale version (ref. 17); we use exclusively the field-scale version here. This version of the model can accommodate very large field or pasture areas (up to hundreds of hectares) or very small areas (minimum of one meter square). In the field-scale version, the plant community of up to seven plant species or functional groups covers the field without spatial constraints.

The seven plant species are distinguished one from another by the 37 physiological and phenological parameters the user inputs to the model. These parameters are a significant feature of the model because the simulation of all the species uses the same code with no branching for different functional groups such as grasses, forbs, or shrubs. The same is true for the soil moisture components of the model. Each of up to nine sites on the field can have up to nine soil layers (minimum of four). Soil characteristics such as porosity, water holding capacity at different moisture tensions, and layer depth serve to distinguish the various layers. The same sections of code are used to simulate each layer in the soil profile.

Though a large number of parameters are needed to initialize the plant component of the model, only about six per plant species need to be input with any degree of accuracy (ref. 30). Thus, simulations can be executed for many locations or situations with the parameters given in Skiles (ref. 31).

Figure 5 shows the control loop structure for SPUR. The execution path for assimilation begins with the year loop, then moves to the month, day, site/field, and soil-layer loop, respectively. Thus, each inner loop is executed before the next outer loop is incremented; all soil layer calculations are done before the site/field loop counter is increased, for example. These important characteristics affect the analysis and restructuring of the control loops in the original code for export to supercomputers.

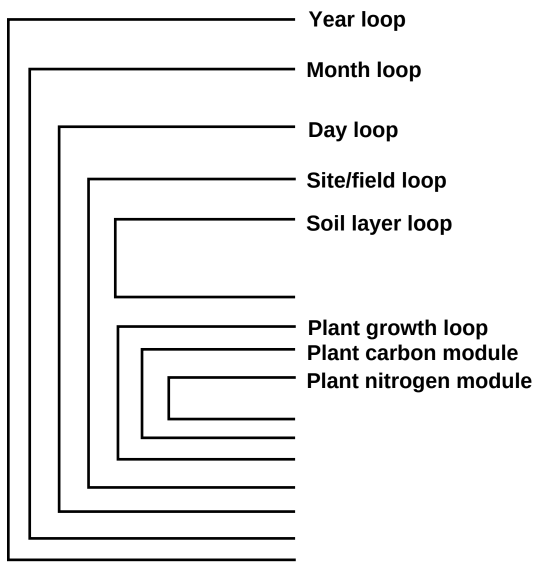

Figure 5. Control loop structure for the SPUR model. Inner loops are executed before outer loops.

Characteristics of the Simulated Site

In this exercise, we simulate a shortgrass prairie site called Pawnee, located approximately 60 km northeast of Fort Collins, Colorado, at an elevation of 1650 m above mean sea level. The Pawnee watershed, part of the Central Plains Experimental Range (CPER) is approximately 40° north latitude and 104° west longitude.

Average precipitation at the site is about 305 mm per year (ref. 1); about 70 percent falls during the May–September growing season (ref. 32). Average wind speed, yearlong, is about 10 km/hr. Soil parameters for our simulations were defined for an Ascolon sandy-loam soil profile.

The vegetation is dominated by warm-season, shortgrass bunch and sod-forming C_4 plants. Over the long term, about 700 kg/ha/yr are produced, but the production may be 50 percent of that in dry years and 250 percent of that in wet years (ref. 33). Other components of the vegetation community include cool-season grasses and forbs, shrubs and half-shrubs, and cactus (ref. 33). See Appendix 1 for estimates of production and plant species at the Pawnee site.

For our work, we configured the SPUR model to include five functional plant groups characteristic of the shortgrass steppe at Pawnee. These were warm season grasses, cool season grasses, warm season forbs, cool season forbs, and shrubs. Parameter values for these functional groups were obtained from Skiles (ref. 31) and Hanson (ref. 19).

Baseline Simulations

We generated a series of simulations using the SPUR model that produced results allowing us to compare and evaluate subsequent model output produced by other computing platforms. The SPUR model was developed on a DEC VAX 11/750 in the mid-1980s (ref. 17). We used a VAX 11/785 for our baseline simulations.

A standard run for the SPUR model consisted of simulating one site for one year with the five plant functional groups mentioned above. The site soil profile was configured for four soil layers. The weather drivers used were from an actual weather record for the same location beginning in 1971.

The summed month-end biomass amounts generated by the model are shown in the first line of table 1. These results conform to the monthly trend of the grassland being simulated, as they show the seasonal dynamics of the community (ref. 34). The amounts are within the norm for the Pawnee site's monthly production (see Appendix 1).

To test the portability of the code, we next executed the model with the same configuration on five other processors: a micro-computer with an Intel 80486 processor, a Sun workstation, a Silicon Graphics, Inc. (SGI) workstation, a Cray Y-MP, and a Cray C90. Summed monthly biomass generated from each of these experiments is also shown in table 1. In the five-species simulations, cool-season plants initiated growth in April. The single species simulations used only the warm-season functional group; hence, growth was initiated during the warmer month of June.

As can be seen, the different processors produced different biomass amounts, and in some instances these differences were as much as 50 percent. These cases occurred mostly in the single-species simulations and in the later months of the simulations, so the variations in the biomass differences were small relative to the total amount of biomass produced. The fact that these differences exist at all provides the major reason for writing this paper.

Table 1. Predicted month-end biomass (kg/ha) from the SPUR grassland model executed on the processors with the precision and configuration shown

		Month											
Processor and configuration		Jan.	Feb.	Mar.	Apr.	May	Jun.	Jul.	Aug.	Sept.	Oct.	Nov.	Dec.
VAX 785	5 sp,[a] 1yr	0.0	0.0	0.0	97.3	361.1	787.5	698.3	519.8	265.7	20.0	0.2	0.0
PC486	5 sp, 1yr	0.0	0.0	0.0	96.7	360.8	787.9	699.5	520.9	266.2	19.3	0.2	0.0
Sun, SP[b]	5 sp, 1yr	0.0	0.0	0.0	97.3	357.5	790.1	700.3	520.6	265.1	19.1	0.2	0.0
Sun, DP[c]	5 sp, 1yr	0.0	0.0	0.0	97.3	361.1	787.4	698.3	519.7	264.1	19.8	0.2	0.0
SGI, SP	5 sp, 1yr	0.0	0.0	0.0	97.3	357.5	790.1	700.3	520.6	265.1	19.1	0.2	0.0
Cray Y-MP, SP	5 sp, 1yr	0.0	0.0	0.0	97.3	361.1	787.4	698.3	519.7	284.1	22.6	0.3	0.0
Cray C90, SP	5 sp, 1yr	0.0	0.0	0.0	97.3	361.1	787.4	698.3	519.7	284.1	22.6	0.3	0.0
VAX 785	1 sp, 1yr	0.0	0.0	0.0	0.0	0.0	583.3	707.0	534.5	282.6	11.4	0.1	0.0
Sun, SP	1 sp, 1yr	0.0	0.0	0.0	0.0	0.0	583.3	707.0	534.5	282.6	11.4	0.1	0.0
Sun, DP	1 sp, 1yr	0.0	0.0	0.0	0.0	0.0	583.3	707.0	534.5	282.6	11.4	0.1	0.0
Cray Y-MP, SP	1 sp, 1yr	0.0	0.0	0.0	0.0	0.0	583.3	707.0	534.5	314.3	15.9	0.1	0.0

[a]sp = species.
[b]SP = single precision.
[c]DP = double precision.

Differences in Results and Possible Causes

The results in table 1 may be due to differences between the computer systems in one or more of the following areas: data representation, computer arithmetic, compilers, mathematical libraries, and conversion of data for input and output. When the differences were initially discovered, we first investigated whether the cause was one of the differences between FORTRAN IV and FORTRAN 77 (ref. 35). In particular, we noted that the Cray run produced an overflow on the variable stack. We re-sized the stack, required static variable allocation, and specified that all variables be saved upon exit from subroutines. In the new run, there was no longer a stack overflow, but the Cray results still did not match those of the test case exactly.

We checked whether DO loop processing affected the outcome. In FORTRAN IV, a DO loop is executed at least once, regardless of whether or not the final value of the loop control variable is greater than its initial value. In FORTRAN 77, a DO loop is skipped if the ending value of the loop control variable is less than the initial value. There were no anomalies in the loop values, and specifying at least one trip through each loop did not have any effect.

After finding that the new runs still did not produce results matching the standard run, we examined function subprograms (e.g., BELL and ATANF) individually to see if there were any differences in the values they produced. We determined that, with the same input, the functions produced the same output on different computing platforms. We examined IF tests that might result in different branches being taken (for different growing conditions, not different functional groups). (See Appendix 2 for a description of each SPUR subroutine and function subprogram.) We concentrated on the parts of the program involved in calculating biomass (e.g., subroutines PLGRO, PLANT, PHOTO, and NITE). By examining the number of calls to a physiological-curve generation subroutine BELL on a day-to-day basis, we discovered that, among the machines, the number of calls on certain days differed.

The following portion of a FORTRAN program modified from SPUR compares the value RTEMP (an intermediate variable) to PHYTM2 (a state variable).

```
RTEMP=PHYTM1*P
IF (RTEMP .LT. PHYTM2) . . .
```

The values of RTEMP and PHYTM2 after 166 days of growth indicate they are equal to seven decimal places.

```
PHYTM1=3.115948E+01
P=10.0
RTEMP=3.115948E+02
PHYTM2=3.115948E+02
```

However, while RTEMP and PHYTM2 were equal on the Cray, RTEMP was less than PHYTM2 on a SUN. An examination of the SUN representation of RTEMP and PHYTM2 showed that the difference occurred in the low-order three bits, the actual difference being one bit in the least significant position. Pinpointing the actual calculation causing the difference in results is difficult because the IF test occurs after many calculations. It does point out, however, that unexpected results can occur when using the same code on different computer systems. Although variation is less common in current machines because of the adoption of IEEE Standard 754 (ref. 36) in the mid 1980s, it is especially important to be aware of data representation differences when porting a code between different machines.

Data representation– In trying to understand the differences between machines, it is important to understand how data are represented in computers. Only a few concepts relating to our experiences with the SPUR model are presented here. Interested readers are referred to Goldberg (ref. 37) for a more complete explanation of floating-point arithmetic. Machine-specific information can be found in Cray (refs. 38 and 39), Levy and Eckhouse (ref. 40), and Sun Microsystems, Inc. (ref. 41).

To be represented by a computer, a decimal floating-point number is first converted to a binary number. It is then stored in a computer word with a sign bit and bits representing the exponent and fraction. It is interpreted as:

$$((-1)^{\text{sign}}) \times (2^{\text{exponent-bias}}) \times (0.f) \qquad (1)$$

where f is the fraction.

Following IEEE Standard 754, a single-precision, 32-bit computer word would use the leftmost bit[2] (bit 31) to indicate the sign of the fraction, bits 30-23 to represent the exponent, and bits 22-0 to represent the fraction. The exponent does not use a signed magnitude representation but uses a representation in which a bias is added to the exponent. Instead of allocating a sign bit to the exponent, in addition to a sign bit for the number, the exponent is represented as a positive number. However, the upper half of the exponent range represents positive numbers and the lower half represents negative numbers. The true value of the exponent is determined by subtracting the bias. This representation facilitates the arithmetic process because

[2]Starting from the left, bits are numbered from 31 to 0 inclusive.

non-negative floating-point numbers can be treated as integers for comparison purposes (ref. 37).

Twenty-three bits are used to represent the fraction. However, the IEEE standard and other representations assume the high-order bit of the fraction is one when the number is normalized, and so they do not represent it. This hidden bit effectively gives 24 bits for representing the fraction. Thus, the bits of the fraction form a binary number as follows:

$$(b_0)(2^{-1}) + (b_1)(2^{-2}) + (b_2)(2^{-3}) + \ldots + (b_{23})(2^{-24}) \qquad (2)$$

where b_n represents the n^{th} bit of the fraction. The value for b_0 is 1 if there is a hidden bit. The Digital Equipment Corporation VAX architecture assumes the binary point is to the left of the most significant (hidden) bit. IEEE assumes the binary point is to the right of the most significant (hidden) bit. Cray does not assume a hidden bit. Table 2 shows how the Cray and VAX compare to IEEE Standard 754 in the representation of single-precision floating-point numbers. When different representations are used for floating-point numbers, there are resulting differences in range and accuracy.

Computer arithmetic– In addition to differences in data representation, differences in computer arithmetic may also play a role in producing different results on different systems. These differences in computer arithmetic are much harder to determine because the use of compilers and mathematical libraries is also involved. Fortunately, IEEE Standard 754 establishes guidelines for computer arithmetic as well as data representation. However, some machine architectures (e.g., Cray and VAX) predate the standard, so it is important to understand the standard, and then to understand how machines deviate from the standard.

Goldberg (ref. 37) addresses the first issue by providing an excellent tutorial on the details of floating-point arithmetic and the IEEE Standard. Machine deviation from the standard is harder to address because the information on older machines may be proprietary and/or obscured by the role of the compiler or mathematical libraries. To address this problem, several programs are available for determining a machine's compliance with the IEEE Standard. Press (ref. 42) provides routines for diagnosing machine parameters. Another tool useful in determining characteristics of computer arithmetic is a program called

Table 2. Differences in single precision floating-point data representation on three computing platforms

Machine	VAX	IEEE Std. 754	Cray
Sign	bit 15	bit 31	bit 63
Exponent	bits 14–7	bits 30–23	bits 62–48
Number of bits	8	8	15
Bias	128	127	2048
Fraction	bits 6–0, 31–16	bits 22–0	bits 47–0
Number of bits	23	23	48
Hidden bit	YES	YES	NO
Effective number of bits	24	24	48
Approximate range			
Maximum	1.7E + 38	3.4E + 38	2.73E + 2465
Minimum	2.9E – 37	1.175E – 38	3.67E – 2466

"Paranoia".[3] The program checks for adherence to IEEE Standard 754 by actually performing arithmetic and comparing the results to those expected. We ran Paranoia on the Cray, VAX, and IEEE-compliant machines and confirmed that there are indeed differences in arithmetic. Appendix 3 contains output from an IEEE-compliant machine, the Cray, and the VAX.

The output from Paranoia identifies a flaw in the VAX arithmetic and serious defects in the Cray arithmetic. These flaws and defects indicate deviations from the IEEE standard and are not indicators that the machines should be dismissed as viable computing platforms. However, a machine's floating point characteristics are of concern to numerical analysts. Because of the potential impact on some numerical algorithms, users should be aware of machine's floating point characteristics .

Resolution of Discrepancies

From our investigations into the reasons for differences in biomass amounts calculated by the various computing platforms, we concluded that the discrepancies were related to how the various machines perform computer arithmetic and not to program errors or differences between FORTRAN IV and FORTRAN 77. To date, we have not made any program changes to accommodate the differences.

Code Optimization

The Cray architecture offers opportunities for faster execution time of the SPUR model beyond that resulting from faster clock speeds. We use the term "optimization" for this process of improving execution times by changing the code configuration. A schematic diagram for the optimization process is shown in figure 6.

Optimization Methodology

To be effective, any optimization scheme must have a goal or an end point. We could have chosen as our goal the conformation of Cray output with VAX results; we could have tried to match exactly the biomass production figures for Pawnee given in Appendix 1. Instead, we chose as our goal decreased CPU or execution time while still producing the numbers shown in our initial testing (table 1). Establishing this criterion is box A in figure 6.

[3]Paranoia can be obtained by sending the e-mail message "send index" to the Internet address netlib@ornl.gov.

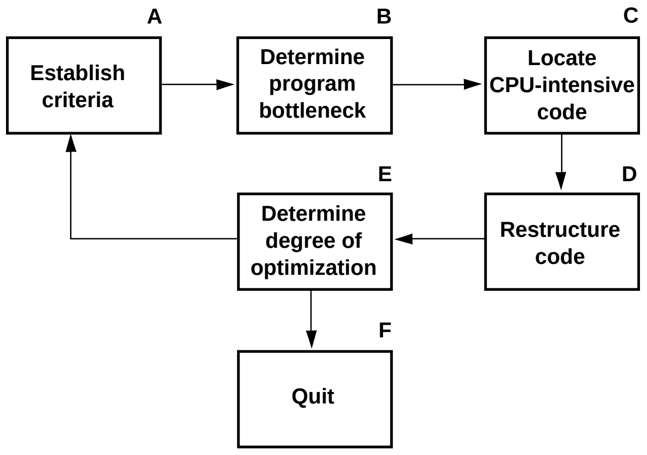
A
Establish criteria
B
Determine program bottleneck
C
Locate CPU-intensive code
D
Restructure code
E
Determine degree of optimization
F
Quit

Figure 6. Flowchart for optimization of code being ported to the Cray. Letters above boxes refer to steps in the process. See text, pages 11–15, for details.

Establish Criteria

Table 3 shows CPU times for the standard run of the SPUR model: simulating one site with five plant functional groups for one year (see Baseline Simulation section). Timings are approximate; runs were done in an environment where the machines were used by others during the timing runs. The C90 standard run is about twenty-eight times faster than the VAX standard run and about ten times faster than the SUN standard run. These results are consistent with the 30-fold speedup to be expected in porting a scalar code to a vector processor machine (ref. 43). We would like to see a speedup of 100 or 200 over a VAX. The first step towards a significant speedup is to determine the program bottleneck.

Determine Program Bottleneck

Cray computer systems offer a suite of tools to help locate CPU-intensive and I/O-intensive portions of code (ref. 44). One of the tools we used was the Hardware Performance Monitor (HPM). HPM is a hardware monitor introducing very little overhead when used in conjunction

Table 3. Comparison of execution times for the standard run of the SPUR model

Machine	CPU time
VAX 11/785	11.95 sec
SUN 4 (33 MHz)	4.34 sec
Cray C90 (1 processor)	0.43 sec

with a program. Table 4 shows a portion of the output HPM produced during a SPUR simulation that was done for one site, five functional groups, and 100 years. This shows that SPUR operates at approximately 83 MIPS (million instructions per second) and 18.8 MFLOPS. The MIPS value is within the normal range of 20–250 MIPS, but the MFLOPS value should be between 30 and 1000 on the C90 (ref. 44). The relatively high MIPS number, the very low number of vector floating-point operations, and the low MFLOPS number indicate that the unmodified SPUR model is clearly not using the full capabilities of the machine.

Table 4. Portion of HPM output for a SPUR simulation

Operation	Number
Million inst/sec (MIPS)	82.96
Floating ops/sec	18.81M
Vector floating ops/sec	0.13M

The higher the number of floating-point operations, the more vectorized and efficient the code. The low number for the unmodified SPUR shows that it is a highly sequential code. This is inherent in the application because it was originally developed for a sequential machine, the DEC VAX. The ultimate success in speeding up SPUR depends on how much of the code is vectorizable (or can be made vectorizable) and how much is sequential. This ratio is important in determining how close an application can come to the peak speed of the machine, as Amdahl's law shows.

Amdahl's Law– Amdahl's Law (ref. 45) is useful in understanding why most applications executed on SIMD or MIMD machines seldom achieve the peak speed of the machine. For vector/pipelined processors such as the Cray, it is important to consider the fraction of the code that is vectorizable. Figure 7 shows the fraction of the code that must be vectorized to achieve a given speedup, given different amounts of machine parallelism. On vector/pipelined machines, machine parallelism equals the vector speed divided by the scalar speed. Speedup is the inverse of the sum of the fraction of unvectorized code and the fraction of vectorized code divided by the machine parallelism. As machine parallelism increases, a larger fraction of the code must be parallelized to achieve a given efficiency. On the Cray C90, the ratio of vector speed to scalar speed is approximately 10–20 (ref. 46). For a ratio of 10, 90 percent vectorization of the code would result in a speedup of approximately 5. (See ref. 47 for comments on limitations and applications of Amdahl's Law.) Unless the ecosystem simulation code can be highly vectorized, the speed on the supercomputer will not be a function of the speed of the vector units, but will instead be a function of the speed of the scalar units.

Locate CPU-intensive Code

To locate CPU-intensive code, we again used Cray utilities to find the subprograms in SPUR that use the most CPU time (box C, fig. 6). Table 5 shows the top twelve

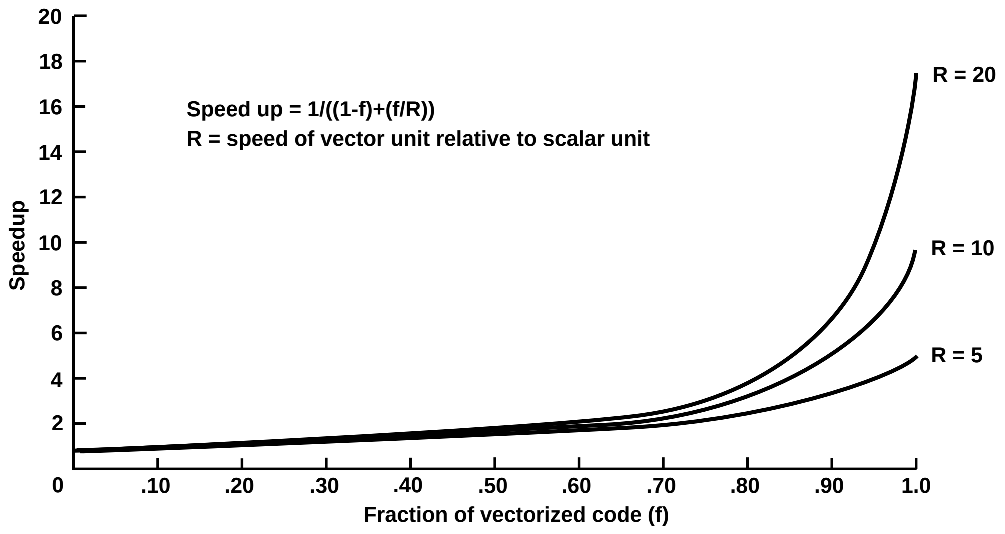

Figure 7. Graphical representation of Amdahl's Law applied to vector processors.

Table 5. Example of Flow Trace Statistics showing routines sorted by descending accumulated CPU time (seconds)

Routine name	Total time	Number of calls	Avg. time/call	Percentage	Acccum. percent
BELL	6.42E-01	89,076	7.21E-06	15.83	15.83
DETAIL	6.29E-01	1,826	3.44E-04	15.50	31.34
TEMPP	4.97E-01	112,468	4.42E-06	12.26	43.60
PEXP	4.90E-01	68,016	7.20E-06	12.08	55.68
PHOTO	4.35E-01	8,502	5.12E-05	10.74	66.41
PLGRO	4.09E-01	9,130	4.48E-05	10.10	76.51
PLANT	1.87E-01	1,826	1.03E-04	4.62	81.13
FSVPI	1.57E-01	1	1.57E-01	3.88	85.01
THRESH	1.31E-01	22,886	5.71E-06	3.22	88.23
HYP	1.14E-01	43,213	2.63E-06	2.80	91.03
NITE	1.02E-01	10,956	9.29E-06	2.51	93.54
SOIL	5.49E-02	1,826	3.00E-05	1.35	94.90

routines in terms of CPU time, sorted in descending order. As shown in table 5, subprograms BELL, DETAIL, TEMPP, and PEXP account for more than 55 percent of the execution time for the standard run of SPUR. Of these four subprograms, DETAIL, which produces daily output, is the only one that writes to the output files. The other three subprograms call intrinsic functions repeatedly and thereby use extensive amounts of CPU time.

The SPUR model was written so that frequently used sections of code are placed in subroutine or function subprograms. On many scalar machines, this results in certain efficiencies. Among them are reducing the length of the code and decreasing the execution time because extensive (conditional) branching is avoided.

Using Cray compiler options, it is possible to place subprograms within the body of the calling program without rewriting the code. This, in effect, restructures the code (box D, fig. 6). This is called the “in-line” feature. The Cray memory is very large relative to scalar machines on which the code was developed, and the efficiencies for scalar computers actually slow down the speed of execution. Using the in-line command at compile time, a function or subroutine subprogram can be placed inside of the main or calling program. Less time is used by the Cray moving between modules, and execution time for a section of code is decreased. In the following conceptual example of in-line restructuring of the program code, the first program block contains a main (calling) program and a subroutine MODULE called by MAIN.

```
PROGRAM MAIN
DIMENSION
COMMON B,C,X
      .
      .
CALL MODULE (A)
      .
      .
END
SUBROUTINE MODULE (A)
COMMON B,C,X
DIMENSION X(10000)
A = VALUE
DO I=1,10000
  A = A + B + C * X(I)
END DO
RETURN
END
```

The second program block shows the result of the in-line command with the pertinent parts of MODULE placed in MAIN.

```
PROGRAM MAIN
DIMENSION
COMMON B,C,X
      .
      .
A = VALUE
DO I=1,10000
  A = A + B + C * X(I)
END DO
      .
      .
END
```

Note that no rewriting of the code is necessary because the Cray compiling systems restructure the code. Table 6 shows routines with an in-line factor of one or greater, sorted by in-line factor. An in-line factor greater than 1 indicates that a routine may qualify for in-lining.

Vectorization information is also available from the Cray. Figure 8 shows a sample of the Cray vectorization output. The system indicates if loops were or were not vectorized. If a loop was not vectorized, information is provided as to why vectorization does not occur. In the example, the loop beginning at line 116 was not vectorized because it contains an inner loop, and only the innermost loops are vectorized. The inner loop beginning at line 117 was vectorized. We used the vectorization information to identify possibilities for additional vectorization.

Restructure Code

To restructure the code, we first placed a number of routines in-line, adding candidate routines fitting the Cray's requirements for in-lining. For example, routines placed in-line should generally have less than 50 lines of code. We also examined the vectorization information to determine if loops should be restructured. Very few routines were actually structured for vectorization. We did not restructure the loops of this program because the loop control variables were similarly-sized, with actual value depending on user input.

We examined the possibility of unrolling loops, removing nested IF statements, assigned GOTO's, backward transfers within loops, and recursion as possible methods to speedup code (ref. 48). We determined that significantly improving vectorization of the SPUR model would mean a substantial reworking of the loops controlling the simulated time and location of the simulation (fig. 5).

Determine Degree of Optimization

As seen in table 7, by placing subroutine and function subprograms in-line, our restructuring of the SPUR model resulted in a 30 percent speedup of execution for a 100 year run over the unmodified version of SPUR. We achieved this speedup on a single processor of the Cray C90. We ran the model for 100 years so that the execution times would be larger; we ran the model for one site and then for nine sites. To ascertain if we could improve vectorization by more fully using the allocated arrays, we ran a nine-site case; however, the number of MFLOPS did not increase much over the one-site case.This net increase did not give us the factor of 100–200 we desired because the amount of vectorization was not substantially increased (the average vector length was between 3 and 12, not close to the optimum of 128). Since our simple restructuring was not enough to increase the vector length, substantial reworking will be needed to improve the code execution.

Table 6. Example of Flow Trace Statistics showing routines sorted by descending in-line factor

Routine name	Total time	Number of calls	Avg. time/call	Percentage	Acccum. percent	In-line factor
TEMPP	4.97E-01	112,468	4.42E-06	12.26	12.26	105.99
HYP	1.14E-01	43,213	2.63E-06	2.80	15.07	68.45
BELL	6.42E-01	89,076	7.21E-06	15.83	30.90	51.49
PEXP	4.90E-01	68,016	7.20E-06	12.08	42.98	39.36
THRESH	1.31E-01	22,886	5.71E-06	3.22	46.20	16.71
ATANF	2.89E-02	9,130	3.16E-06	0.71	46.91	12.02
CRACK	2.51E-02	7,304	3.44E-06	0.62	47.53	8.86
NITE	1.02E-01	10,956	9.29E-06	2.51	50.04	4.92
ALBEDO	5.78E-03	1,826	3.17E-06	0.14	50.18	2.40
FLDHYD	1.00E-02	1,826	5.50E-06	0.25	50.43	1.38
DAYREP	1.27E-02	1,826	6.96E-06	0.31	50.74	1.09
EVAPR	1.33E-02	1,826	7.26E-06	0.33	51.07	1.05
PHOPER	1.39E-02	1,826	7.60E-06	0.34	51.41	1.00

```
1882  116. S S S S---------<      DO 60 NC2 = 1,NSITE
1883  117. S S S S V-------<      DO 60 L = 1,NCP
1884  118. S S S S-V------->   60 IF(FIX(NC2,L) .EQ. 0.0)SM = SM + Q2(NC2,L)
1885  119. S S S S---------<      DO 70 NC2 = 1,NSITE
1886  120. S S S S V-------<      DO 70 L = 1,NCP
1887  121. S S S S-V------->   70 IF(FIX(NC2,L) .EQ. 0.0)Q2(NC2,L) =Q2(NC2,L) + Q2(NC2,L)
1888  122. S S S                 +                        * X/SM
1888  123. S S S                  SUM = 1.0
1889  124. S S S                  IT = IT + 1
1890  125. S-S-S----------->      IF(IT .LE.( NCP * NSITE + 1))GO TO40
1891  126.   S-S----------->   80 CONTINUE
```

```
cft77-8035 cf77: VECTOR NTRFC, Line = 116, File = spur2.f, Line = 1882
  Loop starting at line 116 was not vectorized.  It contains an inner loop.
cft77-8004 cf77: VECTOR NTRFC, Line = 117, File = spur2.f, Line = 1883
  Loop starting at line 117 was vectorized.
cft77-8035 cf77: VECTOR NTRFC, Line = 119, File = spur2.f, Line = 1885
  Loop starting at line 119 was not vectorized.  It contains an inner loop.
cft77-8004 cf77: VECTOR NTRFC, Line = 120, File = spur2.f, Line = 1886
  Loop starting at line 120 was vectorized.
```

Figure 8. Sample vectorization information.

Table 7. The degree of optimization achieved using the in-line tool for 100-year model runs using the Cray C90 (1 CPU). The subroutines placed in-line were `BELL`, `PEXP`, `TEMPP`, `THRESH`, `HYP`, `ATANF`, `CRACK`, `ALBEDO`, and `PHOPER`

	CPU-time, sec	FLOPS	Vector FLOPS	Percent speedup
5 species, 1 site				
SPUR	32.30	18.81M	0.13M	
SPUR w/in-line	25.26	22.91M	0.44M	27.87
5 species, 9 sites				
SPUR	226.54	19.46M	0.11M	
SPUR w/in-line	175.30	24.00M	0.50M	29.23

Future Plans

We believe that SPUR and models like SPUR would be more useful if larger versions with increased dimensions could be run. A thousand-time speedup of the SPUR model execution was not realized because code written for use on scalar machines does not take advantage of the specialized functional units of a supercomputer (ref. 43). Our efforts so far reveal that substantial restructuring of the code will be needed, and that merely porting the code to a new platform is insufficient. We plan to vectorize and parallelize the SPUR model to gain this increase.

Figure 5 shows the control loop structure for the SPUR model. The site calculations can be done independently so that up to nine sites can be evaluated simultaneously. In addition, the plant growth loop calculations can be done independently for each plant species so that up to seven species can be evaluated independently. By parallelizing these calculations, it should be possible to evaluate a 7-species, 9-site model in the same time as a 1-species, 1-site model. Executions of the current version of the code indicate that parallelizing could result in a 6-times speedup in execution time. We predict that even better improvements are possible in spite of overhead introduced by the parallelization. The restructuring would also improve input/output speed, increase vectorization, and remove obsolete coding elements. In addition, we plan to modernize the code by integrating the model into a distributed heterogeneous computing environment; producing two-dimensional and three-dimensional graphical output; and modifying the model to use input from remote-sensing databases and/or real time sensor information.

Summary and Conclusions

Many computer codes simulating ecosystems and ecosystem processes have been developed over the last two decades (ref. 49). The computers used to build and test these models have generally been those available at the time of development. However, the use of supercomputers in ecosystem simulation has been small because these machines are not readily available and are difficult to learn to use. In addition, according to Thromborson (ref. 43), many codes only achieve a 30-fold speedup that does not make supercomputer use cost-effective. Our results are consistent with this figure, but we differ with Thomborson's contention that codes originally developed on scalar machines are not worth the time to modify for execution on supercomputers. We believe that many ecosystem model codes would be more useful if larger models could be run in a shorter period of time, allowing more alternatives to be simulated and more complex questions to be answered. Supercomputers can immediately accomplish these objectives, and in the long-term will enable more effective execution on the workstations that may eventually replace them.

Supercomputers may not be as cost-effective as workstations in some cases. However, if supercomputers are available, they offer the opportunity to begin modernizing codes originally developed on scalar machines such as the VAX. The definition of what constitutes a supercomputer changes over time; capabilities of a supercomputer today may be available on the desk top within a decade (ref. 50). Further, as shown in figure 1, the definition of a supercomputer changes along with the requirements of the user community and the increased power of the machine. What is defined as a supercomputer today may appear as a desktop computer tomorrow, complete with the vectorization and parallelization capabilities previously limited to supercomputers. For codes to run well on the future desktop, they will have to be restructured in the same ways as they now have to be restructured for supercomputers.

To begin a major restructuring effort, the user should have a stable model and a reference or standard run available to compare output between results generated on a supercomputer and results produced by the machine used to develop the code. Because of possible differences in computer arithmetic between platforms, the user should be aware that results produced on a supercomputer can vary from those produced by the reference machine. Consequently, users should establish the amount of difference they are willing to accept between the results from the supercomputer execution and the standard run.

The supercomputer vendor often supplies tools and techniques to optimize computer code. It is important that the user be aware of these tools and learn to apply them where appropriate. In our experience, the in-line commands reduced the execution time of our code. This technique is fully documented in the manuals supplied by the vendor.

Other more direct techniques for computer code optimization may be found in Bently (ref. 51), and a further discussion of ecosystem code optimization may be found in Loehle (ref. 52). These techniques include writing the code to avoid double precision (on the Cray, the word length is large, so this is generally not a problem); using reciprocal multiplication instead of division; making the shorter loops the outer loops in nested tasks, and minimizing the I/O in the code. In the SPUR model, reports are generated from information stored in scratch files and COMMON blocks and written from one subroutine. The user can turn on write statements in other parts of the code with switch options at execution time, but our experience indicates this does not measurably increase execution time on the Cray.

It is important for ecosystem modelers to use tools at hand now. These are the supercomputer-class machines, among them the Cray C90. New and innovative ways of using existing workstations in networks so that they have much of the speed and other resources of a supercomputers are being explored (ref. 53). Programming and control languages are being designed so that by using these techniques much of the coding of models is inherently vectorized during the model construction (ref. 54). This will lead to ecosystem simulation codes ready for use on supercomputing platforms from their inception.

Meanwhile, the modification and execution of ecosystem simulation codes on supercomputers can be realized, enabling more complex systems to be simulated and increasingly complex questions to be asked of ecosystem simulation codes originally constructed on scalar computers.

References

1. Innis, G. S., ed.: Grassland Simulation Model. Ecol. Stud., vol. 26, Springer-Verlag, 1977.

2. Urban, D. L.; and O'Neill, R. V.: Linkages in Hierarchical Models. Coupling of Ecological Studies with Remote Sensing: Potential at Four Biosphere Reserves in the United States, M. I. Dyer and D. A Crossley, Jr., eds., U. S. Department of State Publication 9504, 1986.

3. Stockwell, D. R. B.; and Green, D. G.: Parallel Computing in Ecological Simulation. Math. and Comput. in Simulation, vol. 32, nos. 1–2, 1990, pp. 249–254.

4. Costanza, R.; and Maxwell, T.: Spatial Ecosystem Modelling Using Parallel Processors. Ecol. Model., vol. 58, nos. 1–4, 1991, pp. 159–183.

5. Sud, Y. C.; Sellers, P. J.; Mintz, Y., et al.: Influence of the Biosphere on the Global Circulation and Hydrological Cycle – a GCM Simulation Experiment. Agric. For. Meteorol., vol. 52, nos. 1–2, 1992, pp. 133–180.

6. Gates, W. L.: The Use of General Circulation Models in the Analysis of the Ecosystem Impacts of Climatic Change. Clim. Change, vol. 7, no. 3, 1985, pp. 267–284.

7. Office of Science and Technology Policy: Grand Challenges 1993: High Performance Computing and Communications. A Report by the Committee on Physical, Mathematical, and Engineering Sciences to Supplement the President's Fiscal Year 1993 Budget, Jan. 1992.

8. Potter, C.; Randerson, J. T.; Field, C. B., et al.: Terrestrial Ecosystem Production: A Process Model Based on Global Satellite and Surface Data. Global Biogeochem. Cycles, vol. 7, no. 44, 1993, pp. 811–841.

9. Dixon, D. A.; and Raveché, H. J.: A National Computing Initiative: A Summary. Fut. Gen. Comp. Sys., vol. 5, nos. 2–3, 1989, pp. 339–345.

10. Levin, S. A.; Moloney, K; Buttel, L.; and Castillo-Chavez, C.: Dynamical Models of Ecosystems and Epidemics. Fut. Gen. Comp. Sys., vol. 5, nos. 2–3, 1989, pp. 265–274.

11. Bretherton, F. P.: The Earth System. Future Generation Computer Systems, vol. 5, nos. 2–3, 1989, pp. 259–264.

12. Burks, A. W.; Goldstine, H. H.; and von Neumann, J.: Preliminary Discussion of the Logical Design of an Electronic Computing Instrument. Datamation, vol. 8, nos. 9–10, 1962, pp. 24–30; pp. 36–41.

13. Hockney, R. W.; and Jesshope, C. R.: Parallel Computers: Architecture, Programming, and Algorithms. Adam Hilger Ltd. (Bristol), 1981.

14. Hockney, R. W.; and Jesshope, C. R.: Parallel Computers 2: Architecture, Programming, and Algorithms. Adam Hilger (Bristol and Philadelphia), 1988.

15. Hwang, K.: Advanced Computer Architecture: Parallelism, Scalability, Programmability. McGraw-Hill, Inc., 1993.

16. Flynn, M. J.: Very High-Speed Computing Systems. IEEE, vol. 54, no. 12, 1966, pp. 1901–1909.

17, Wight, J. R.; and Skiles, J. W., eds.: SPUR – Simulation of Production and Utilization of Rangelands: Documentation and User Guide. U.S. Department of Agriculture, Agricultural Research Service, ARS-63, 1987.

18. Springer, E. P.; Johnson, C. W.; Cooley, K. R., et al.: Testing the SPUR Hydrology Component on Rangeland Watersheds in Southwest Idaho. Trans. Am. Soc. Agric. Engrs., vol. 27, 1984, pp. 1040–1046; p. 1054.

19. Hanson, J. D.; Skiles, J. W.; and Parton, W. J.: A Multispecies Model for Rangeland Plant Communities. Ecol. Model., vol. 44, nos. 1–2, 1988, pp. 89–123.

20. Richardson, C. W.; Hanson, C. L.; and Huber, A. L.: Climate Generator. SPUR – Simulation of Production and Utilization of Rangelands: Documentation and User Guide, J. R. Wight and J. W. Skiles, eds., U.S. Department of Agriculture, Agricultural Research Service, ARS-63, 1987, pp. 3–16.

21. Renard, K. G.; Shirley, E. D.; Williams, J. R., et al.: SPUR Hydrology Component: Upland Phases. SPUR – Simulation of Production and Utilization of Rangelands: A Rangeland Model for Management and Research, J. R. Wight, ed., U. S. Department of Agriculture, Misc. Pub. no. 1431, 1983, pp. 17–44.

22. Cooley, K. R.; Springer, E. P.; and Huber, A. L.: Hydrology Component: Snowmelt. SPUR – Simulation of Production and Utilization of Rangelands: A Rangeland Model for Management and Research, J. R. Wight, ed., U. S. Department of Agriculture, Misc. Pub. no. 1431, 1983, pp. 45–61.

23. Skiles, J. W.; Hanson, J. D.; and Parton, W. J.: Simulation of Above- and Below-Ground Carbon and Nitrogen Dynamics of *Bouteloua gracilis* and *Agropyron smithii*. Analysis of Ecological Systems: State-of-the-Art in Ecological Modelling, W. K. Lauenroth; G. V. Skogerboe; and M. Flug, eds., Proceedings, Developments in Environmental Modelling: 5, Colorado State Univ., May 24–28, 1982.

24. Stout, W. L.; Vona-Davis, L. C.; Skiles, J. W., et al.: Evaluating SPUR Model for Predicting Animal Gains and Biomass on Eastern Hill Land Pastures. Agric. Syst., vol. 34, 1990, pp. 169–178.

25. Field, L. B.: Simulation of Beef-Heifer Production on Rangeland. M. S. Thesis, Dept. of Animal Science, Colorado State Univ., 1987.

26. Baker, B. B.; Bourdon, R. M.; and Hanson, J. D.: FORAGE: A Model of Forage Intake in Beef Cattle. Ecol. Model., vol. 60, 1992, pp. 257–279.

27. Sasowsky, K. C.; and Gardner, T. W.: Watershed Configuration and Geographic Information System Parameterization for SPUR Model Hydrologic Simulations. Water Resour. Bull., vol. 27, 1991, pp. 7–17.

28. Hanson, J. D.; Baker, B. B.; and Bourdon, R. M.: Comparison of the Effects of Different Climate Change Scenarios on Rangeland Livestock Production. Agric. Syst., vol. 41, no. 4, 1993, pp. 487–502.

29. Skiles, J. W.; and Hanson, J. D.: Response of Arid and Semiarid Watersheds to Increasing Carbon Dioxide and Climate Change as Shown by Simulation Studies. Clim. Change, vol. 26, no. 4, 1994, pp. 377–397.

30. MacNeil, M. D.; Skiles, J. W.; and Hanson, J. D.: Sensitivity Analysis of a General Rangeland Model. Ecol. Model., vol. 29, 1985, pp. 57–76.

31. Skiles, J. W.: Sample Data Sets for the Field-Scale Version. SPUR – Simulation of Production and Utilization of Rangelands: Documentation and User Guide, J. R. Wight and J. W. Skiles, eds., U. S. Department of Agriculture, Agricultural Research Service, ARS-63, 1987, pp. 321–337.

32. Parton, W. J.; Lauenroth, W. K.; and Smith, F. M.: Water Loss from a Shortgrass Steppe in Northeastern Colorado. Agr. Meteor., vol. 24, 1981, pp. 97–109.

33. Sims, P. L.; Singh, J. S.; and Lauenroth, W. K.: The Structure and Function of Ten Western North American Grasslands: I. Abiotic and Vegetational Characteristics. J. Ecol., vol. 66, 1978, pp. 251–285.

34. Sims, P. L.; and Singh, J. S.: The Structure and Function of Ten Western North American Grasslands: II. Intra-Seasonal Dynamics in Primary Producer Compartments. J. Ecol., vol. 66, 1978a, pp. 547–572.

35. American National Standard Programming Language FORTRAN. ANSI X3.9–1978, American National Standards Institute, Inc., 1978.

36. ANSI/IEEE Standard 754–1985 for Binary Floating-Point Arithmetic. IEEE, Inc., 1985.

37. Goldberg, D.: What Every Computer Scientist Should Know about Floating-Point Arithmetic. ACM Computing Surveys, vol. 23, no. 1, 1991, pp. 5–48.

38. Cray X-MP Computer Systems, Cray X-MP Series Mainframe Reference Manual. HR-0032, Cray Research, Inc., Mendota Heights, Minn., 1982.

39. CF77 Compiling System, Volume 1: FORTRAN Reference Manual. SR-30715.0, Cray Research, Inc., Mendota Heights, Minn., 1990.

40. Levy, H. M.; and Eckhouse, R. H., Jr.: Computer Programming and Architecture: The VAX – II. Digital Press, Bedford, Mass., 1980.

41. Numerical Computation Guide. 800-3555-10, Sun Microsystems, Inc., Mountain View, Calif., 1990.

42. Press, W. H.; Teukolsky, S. A.; Flannery, B. P.; and Vetterling, W. T.: Numerical Recipes: The Art of Scientific Computing (FORTRAN Version). Second ed., Cambridge University Press, 1993.

43. Thomborson, C. D.: Does Your Workstation Computation Belong on a Vector Supercomputer? Comm. of the ACM, vol. 36, no. 11, 1993, pp. 41–49.

44. Unicos Performance Utilities Reference Manual. SR-2040/7.0, Cray Research Inc., Mendota Heights, Minn., 1992.

45. Amdahl, G. M.: The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conference Proceedings, vol. 30, 1967, pp. 483–485.

46. CF77 Optimization Guide. SG-3773/6.0, Cray Research, Inc., Mendota Heights, Minn., 1993.

47. Hillis, W. D.; and Boghosian, B. M.: Parallel Scientific Computation. Science, vol. 261, 1993, pp. 856–863.

48. Levesque, J. M.; and Williamson, J. W.: A Guidebook to FORTRAN on Supercomputers. Academic Press, 1989.

49. Ågren, G. I.; McMurtrie, R. E.; Parton, W. J., et al.: State-of-the-Art of Models of Production-Decomposition Linkages in Conifer and Grassland Ecosystems. Ecological Applications, vol. 1, no. 2, 1991, pp. 118–138.

50. Baskett, F.; and Hennessy, J. L.: Microprocessors: From Desktops to Supercomputers. Science, vol. 261, no. 5123, 1993, pp. 864–871.

51. Bently, J. L.: Writing Efficient Programs. Prentice-Hall, 1982.

52. Loehle, C.: Optimizing Ecosystem Simulation Model Performance. Nat. Resour. Model., vol. 1, no. 2, 1987, pp. 235–243.

53. Buzbee, B.: Workstation Clusters Rise and Shine. Science, vol. 261, no. 5123, 1993, pp. 852–853.

54. Fisher, J. A.; and Rau, B. R.: Instruction-Level Parallel Processing. Science, vol. 253, no. 5025, 1991, pp. 1233–1241.

55. Sims, P. L.; and Singh, J. S.: The Structure and Function of Ten Western North American Grasslands. III. Net Primary Production, Turnover and Efficiencies of Energy Capture and Water Use. J. Ecol., vol. 66, 1978b, pp. 573–597.

56. Dodd, J. L.; and Lauenroth, W. K.: Analysis of the Response of a Grassland Ecosystem to Stress. Perspectives in Grassland Ecology, N. French, ed., Ecol. Stud., vol. 32. Springer-Verlag, 1979, pp. 43–58.

Appendix 1

Pawnee Plant Production

Estimates for biomass and production at the Pawnee location given in this appendix are from Sims et al. (ref. 33), Sims and Singh (refs. 34 and 55), and Innis (ref. 1).

Studies of long-term total aboveground standing crop biomass and of aboveground net primary production show these grazinglands to produce about 700 kg/ha ranging, over a few years, from 50 percent lower to 50 percent higher. Warm-season grasses, the dominant forage plants, have aboveground net primary production of about 400 to 800 kg/ha over a series of a few years. Shrub and half-shrub production may vary from about 200 to 750 kg/ha over a few years' interval.

Over several years, using different treatments (ref. 56), the average standing crop biomass of aboveground vegetation (live and dead material) is about 2590 kg/ha. Some 43 percent of the vegetation comes from succulent species (primarily *Opuntia polyacantha*) and 57 percent from non-succulent plants. Only 1 percent of the standing crop biomass is from legumes, with 99 percent from non-leguminous plants. Some 8 percent of the standing crop biomass is contributed by annual plant species, whereas 92 percent originates from perennial plant species. Of the total standing crop biomass aboveground, 42 percent are contributed by grasses and grasslike plants, 16 percent by forbs (herbaceous, non-gramineous plants), 19 percent by shrubs and half-shrubs, and 42 percent by succulents. These values include both current-year's live, current-year's dead, perennial live, and old dead plant materials.

From a standing crop biomass standpoint, the most important species is *Opuntia polyacantha* with a mean of 1060 kg/ha, followed by *Bouteloua gracillis* with 520 kg/ha and *Artemisia frigida* with 480 kg/ha. Plant species whose standing crop biomass is in the range of 50 to 100 kg/ha include, in order of decreasing importance, *Psoralea tenuiflora*, *Sphaeralcea coccinea*, *Gutierrezia sarothrae*, and *Gura coccinea*. Plant species whose standing crop biomass is in the range of 25 to 50 kg/ha include *Bucloe dactyloides*, *Chrysopsis villosa*, and *Aristida longiseta*. Plant species contributing a significant amount of standing crop biomass (up to 25 kg/ha) include the following in order of decreasing importance: *Carex eleocharis*, *Conyza canadensis*, *Salsola kali*, *Sitanion hystrix*, *Lepidium densiflorum*, *Plantago patagonica*, *Sporobolus cryptandrus*, *Lappula redowski*, and *Orobanche ludoviciana*.

Appendix 2

SPUR Modules

The modules below are for the SPUR Field-Scale Version, Phase I, grassland ecosystem model. Each module is identified as a main program, subroutine subprogram, or function subprogram. The purpose of each module is given, as are names of the modules calling the module and the modules called.

PROGRAM FSVPI (main calling program) calls ALBEDO, DAYREP, DETAIL, ERR, FLDHYD, IOSET, LINE, NDPM, PACK19, PLANT, SOLADJ, TLAPSE, USER, and YRREP, and is called by no subprograms.

The subroutine subprograms and function subprograms below are listed in alphabetical order.

SUBROUTINE ADPL determines the shape of the areal depletion curve and calls no subprograms; ADPL is called buy USER.

SUBROUTINE AESC19 computes the areal extent of snow cover; AESC19 calls no subprograms and is called by PACK19 and USER.

FUNCTION ALBEDO determines albedo for snow covered fields; ALBEDO calls no subprograms and is called by FSVPI.

SUBROUTINE ANIMAL controls execution of the wildlife and livestock subprograms; ANIMAL calls LVSTK and WLDLF, and is called by PLANT.

FUNCTION ATANF calculates plant physiological response based on the arctanget function; ATANF calls no subprograms and is called by PLGRO.

FUNCTION BELL calculates plant physiological response based on a bell-shaped function; BELL calls no subprograms and is called by NITE, PEXP, and PLGRO.

SUBROUTINE CRACK allows part of the water entering a soil layer to seep through the cracks in the layer; CRACK calls no subprograms and is called by SOIL.

SUBROUTINE DAYREP writes daily values of plant biomass and animal weight; DAYREP calls LINE and is called by FSVPI.

SUBROUTINE DETAIL controls output via print switches; DETAIL calls no subprograms and is called by FSVPI, LVSTK, PLANT, and SOILM.

SUBROUTINE ERR reports error codes passed to this subprogram; ERR is called by FSVPI and USER.

SUBROUTINE EVAPR computes plant and soil evaporation; EVAPR calls no subprograms and is called by SOIL.

SUBROUTINE FLDHYD computes surface runoff from a site; FLDHYD calls SOIL and is called by FLDHYD.

SUBROUTINE GROW computes the physiological growth of a steer; GROW calls no subprograms and is called by LVSTK.

FUNCTION HYP calculates plant physiological hyperbolic response curve; HYP calls no subprograms and is called by NITE and PLGRO.

SUBROUTINE IOSET reads data file names and opens appropriate logical unit devices; IOSET calls no subprograms and is called by FSVPI.

SUBROUTINE LINE adjusts page contents for a line printer; LINE calls no subprograms and is called by FSVPI, DAYREP, USER, and YRREP.

SUBROUTINE LVSTK controls livestock routines; LVSTK calls DETAIL, GROW, and NTRFC, and is called by ANIMAL.

SUBROUTINE MELT19 computes surface melt based on 100 percent snow cover and non-rain conditions; MELT19 calls no subprograms and is called by PACK19.

SUBROUTINE NDPM returns the number of days in each month of the current year; NDPM calls no subprograms and is called by FSVPI.

SUBROUTINE NITE calls BELL and HYP and is called by PLGRO.

SUBROUTINE NTRFC interfaces plant and animal components in the model; NTRFC calls ZERO and is called by LVSTK and WLDLF.

SUBROUTINE PACK19 executes snow accumulation and melt module for one computational period; PACK19 calls AESC19, MELT19, ROUT19, and ZERO19 and is called by FSVPI.

SUBROUTINE PERC allows part of the water stored in a soil layer to percolate out of the layer; PERC calls no subprograms and is called by SOIL.

FUNCTION PEXP calculates expected photosynthesis by plant species by photoperiod; PEXP calls BELL and is called by PHOTO.

FUNCTION PHOPER calculates photoperiod of a day based on time of year; PHOPER calls no subprograms and is called by PLANT.

SUBROUTINE PHOTO calculates actual photosynthesis by plant species; PHOTO calls PEXP and TEMPP and is called by PLGRO.

SUBROUTINE PLANT controls plant module components; PLANT calls ANIMAL, DETAIL, PHOPER, PLGRO, SOILM, and TEMPP and is called by FSVPI.

SUBROUTINE PLGRO controls plant growth functions; PLGRO calls ATANF, BELL, HYP, NITE, PHOTO, TEMPP, and THRESH and is called by PLANT.

SUBROUTINE ROUT19 routes excess water through the snow cover; ROUT19 calls no subprograms and is called by PACK19.

SUBROUTINE SOIL distributes evaporation and rainfall excess to the various soil layers; SOIL calls CRACK, EVAPR, and PERC and is called by FLDHYD.

SUBROUTINE SOILC determines the moisture characteristic function for each layer for each site; SOILC calls no subprograms and is called by USER.

SUBROUTINE SOILM calculates soil water potentials for each layer given soil water for each layer; SOILM calls DETAIL and is called by PLANT.

FUNCTION SOLADJ adjusts solar radiation input for slope, aspect, and day of the year; SOLADJ calls no subprograms and is called by FSVPI.

FUNCTION TEMPP calculates soil temperature profile; TEMPP calls no subprograms and is called by PHOTO, PLANT, and PLGRO.

FUNCTION THRESH calculates plant physiological threshold response; THRESH calls no subprograms and is called by PLGRO.

FUNCTION TLAPSE calculates temperature lapse due to altitude; TLAPSE calls no subprograms and is called by FSVPI.

SUBROUTINE USER is the initialization and initial values output subprogram; USER calls ADPL, AESC19, ERR, LINE, and SOILC and is called by FSVPI.

SUBROUTINE WLDLF controls wildlife; WLDLF calls NTRFC and is called by ANIMAL.

SUBROUTINE YRREP writes annual and monthly reports; YRREP calls LINE and is called by FSVPI.

SUBROUTINE ZERO zeros a matrix in the domestic herbivore subroutines; ZERO calls no subprograms and is called by NTRFC.

SUBROUTINE ZERO19 sets all carry-over values to no snow conditions for the snow operation; ZERO19 calls no subprograms and is called by PACK19.

Appendix 3

Paranoia Output

IEEE Standard 754 Machine Output from Run of Paranoia, Single Precision

```
Is this a program restart after failure (1)
or a start from scratch (0) ?
A Paranoid Program to Diagnose Floating-point Arithmetic
... Single-Precision Version ...
Lest this program stop prematurely, i.e. before displaying
 “End of Test”
try to persuade the computer NOT to terminate execution
whenever an error such as Over/Underflow or Division by
Zero occurs, but rather to persevere with a surrogate value
after, perhaps, displaying some warning. If persuasion
avails naught, don’t despair but run this program anyway
to see how many milestones it passes, and then run it
again. It should pick up just beyond the error and
continue. If it does not, it needs further debugging.

Users are invited to help debug and augment this program
so that it will cope with unanticipated and newly found
compilers and arithmetic pathologies.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 0, ... page 1


Please send suggestions and interesting results to
    Richard Karpinski
    Computer Center U-76
    University of California
    San Francisco, CA 94143-0704
    USA

In doing so, please include the following information:
    Precision: Single;
    Version: 31 July 1986;
    Computer:

    Compiler:

     Optimization level:

    Other relevant compiler options:


To continue diagnosis, press return.
Diagnosis resumes after milestone # 1, ... page 2


BASIC version (C) 1983 by Prof. W. M. Kahan.
Translated to FORTRAN by T. Quarles and G. Taylor.
Modified to ANSI 66/ANSI 77 compatible subset by
Daniel Feenberg and David Gay.
You may redistribute this program freely if you
acknowledge the source.
```

```
Running this program should reveal these characteristics:

b = radix ( 1, 2, 4, 8, 10, 16, 100, 256, or ... ) .
p = precision, the number of significant b-digits carried.
u2 = b/b^p = one ulp (unit in the last place) of 1.000xxx..
u1 = 1/b^p = one ulp of numbers a little less than 1.0.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 2, ... page 3

g1, g2, g3 tell whether adequate guard digits are carried;
1 = yes, 0 = no; g1 for mult., g2 for div., g3 for subt.
r1,r2,r3,r4 tell whether arithmetic is rounded or chopped;
0=chopped, 1=correctly rounded, -1=some other rounding;
r1 for mult., r2 for div., r3 for add/subt., r4 for sqrt.
s=1 when a sticky bit is used correctly in rounding; else s=0 .
u0 = an underflow threshold.
e0 and z0 tell whether underflow is abrupt, gradual or fuzzy
v = an overflow threshold, roughly.
v0 tells, roughly, whether infinity is represented.
Comparisons are checked for consistency with subtraction
  and for contamination by pseudo-zeros.
Sqrt is tested. so is y^x for (mostly) integers x .
Extra-precise subexpressions are revealed but not yet tested.
Decimal-binary conversion is not yet tested for accuracy.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 3, ... page 4

The program attempts to discriminate among:
  >FLAWs, like lack of a sticky bit,
  >SERIOUS DEFECTs, like lack of a guard digit, and
  >FAILUREs, like 2+2 = 5 .
Failures may confound subsequent diagnoses.

The diagnostic capabilities of this program go beyond an
earlier program called “Machar”, which can be found at the
end of the book “Software Manual for the Elementary Functions”
(1980) by W. J. Cody and W. Waite. Although both programs
try to discover the radix (b), precision (p) and
range (over/underflow thresholds) of the arithmetic, this
program tries to cope with a wider variety of pathologies
and to say how well the arithmetic is implemented.
The program is based upon a conventional radix
representation for floating-point numbers,
but also allows for logarithmic encoding (b = 1)
as used by certain early wang machines.


To continue diagnosis, press return.
Diagnosis resumes after milestone # 7, ... page 5

Program is now RUNNING tests on small integers:
-1, 0, 1/2 , 1, 2, 3, 4, 5, 9, 27, 32 & 240 are O.K.
```

```
Searching for radix and precision...
Radix = 2.
Closest relative separation found is 5.96046448E-08
Recalculating radix and precision
confirms closest relative separation .
Radix confirmed.
The number of significant digits of radix 2. is 24.00
Test for extra-precise subexpressions:
Subexpressions do not appear to be calculated
with extra precision.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 30, ... page 6

Subtraction appears to be normalized as it should.
Checking for guard digits in multiply divide and subtract.
These operations appear to have guard digits as they should.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 40, ... page 7

Checking for rounding in multiply, divide and add/subtract:
Multiplication appears to be correctly rounded.
Division appears to be correctly rounded.
Add/subtract appears to be correctly rounded.
checking for sticky bit:
Sticky bit appears to be used correctly.

Does multiplication commute? Testing if x*y = y*x for 20 random pairs:
No failure found in 20 randomly chosen pairs.

Running tests of square root...
Testing if sqrt(x*x) = x for 20 integers x.
Found no discrepancies.
Sqrt has passed a test for monotonicity.
Testing whether sqrt is rounded or chopped:
Square root appears to be correctly rounded.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 90, ... page 8

Testing powers z^i for small integers z and i :
Start with 0.**0 .
No discrepancies found.

Seeking underflow threshold and min positive number:
Smallest strictly positive number found is minpos = 1.40129846E-45
Since comparison denies MINPOS = 0,
 evaluating ( MINPOS + MINPOS ) / MINPOS should be safe;
what the machine gets for ( MINPOS + MINPOS ) / MINPOS is
     0.2000000E+01
This is O.K. provided over/underflow has not just been signaled.
Underflow is gradual; it incurs absolute error =
(roundoff in underflow threshold) < minpos.
The underflow threshold is 0.11754945E-37 , below which
calculation may suffer larger relative error than merely roundoff.
```

```
To continue diagnosis, press return.
Diagnosis resumes after milestone # 130, ... page 9

since underflow occurs below the threshold =
     ( 2.00000000E+00)^( -1.26000000E+02) ,
only underflow should afflict the expression
     ( 2.00000000E+00)^( -2.52000000E+02) ;
actually calculating it yields
 0.00000000E+00
This computed value is O.K.
Testing x^((x+1)/(x-1)) vs. exp(2) = 0.73890557E+01 as x -> 1.
Accuracy seems adequate.
Testing powers z^q at four nearly extreme values:
No discrepancies found.


To continue diagnosis, press return.
Diagnosis resumes after milestone # 160, ... page 10

Searching for overflow threshold:
Can “ z = -y “ overflow? trying it on y = Infinity
Seems O.K.
Overflow threshold is v = 3.40282347E+38
Overflow saturates at sat = Infinity
No overflow should be signaled for v*1 =
                   3.40282347E+38
            nor for v/1 =
                   3.40282347E+38
Any overflow signal separating this * from one above is a DEFECT.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 190, ... page 11


What messages and/or values does division by zero produce?
About to compute 1/0...
Trying to compute 1/0 produces Infinity
About to compute 0/0...
Trying to compute 0/0 produces NaN


To continue diagnosis, press return.
Diagnosis resumes after milestone # 220, ... page 12

No failures, defects nor flaws have been discovered.
Rounding appears to conform to the proposed IEEE standard P754
The arithmetic diagnosed appears to be Excellent!
End of Test.
```

Cray Output from Run of Paranoia, Single Precision

```
Is this a program restart after failure (1)
or a start from scratch (0) ?
A Paranoid Program to Diagnose Floating-point Arithmetic
    ... Single-Precision Version ...
Lest this program stop prematurely, i.e. before displaying
 “End of Test”
try to persuade the computer NOT to terminate execution
whenever an error such as Over/Underflow or Division by
Zero occurs, but rather to persevere with a surrogate value
after, perhaps, displaying some warning. If persuasion
avails naught, don’t despair but run this program anyway
to see how many milestones it passes, and then run it
again. It should pick up just beyond the error and
continue. If it does not, it needs further debugging.

Users are invited to help debug and augment this program
so that it will cope with unanticipated and newly found
compilers and arithmetic pathologies.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 0, ... page 1


Please send suggestions and interesting results to
    Richard Karpinski
    Computer Center U-76
    University of California
    San Francisco, CA 94143-0704
    USA

In doing so, please include the following information:
    Precision: Single;
    Version: 31 July 1986;
    Computer:

    Compiler:

     Optimization level:

    Other relevant compiler options:


To continue diagnosis, press return.
Diagnosis resumes after milestone # 1, ... page 2


BASIC version (C) 1983 by Prof. W. M. Kahan.
Translated to FORTRAN by T. Quarles and G. Taylor.
Modified to ANSI 66/ANSI 77 compatible subset by
Daniel Feenberg and David Gay.
You may redistribute this program freely if you
acknowledge the source.
```

```
Running this program should reveal these characteristics:

b = radix ( 1, 2, 4, 8, 10, 16, 100, 256, or ... ) .
p = precision, the number of significant b-digits carried.
u2 = b/b^p = one ulp (unit in the last place) of 1.000xxx..
u1 = 1/b^p = one ulp of numbers a little less than 1.0.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 2, ... page 3

g1, g2, g3 tell whether adequate guard digits are carried;
1 = yes, 0 = no; g1 for mult., g2 for div., g3 for subt.
r1,r2,r3,r4 tell whether arithmetic is rounded or chopped;
0=chopped, 1=correctly rounded, -1=some other rounding;
r1 for mult., r2 for div., r3 for add/subt., r4 for sqrt.
s=1 when a sticky bit is used correctly in rounding; else s=0 .
u0 = an underflow threshold.
e0 and z0 tell whether underflow is abrupt, gradual or fuzzy
v = an overflow threshold, roughly.
v0 tells, roughly, whether infinity is represented.
Comparisons are checked for consistency with subtraction
  and for contamination by pseudo-zeros.
Sqrt is tested. so is y^x for (mostly) integers x .
Extra-precise subexpressions are revealed but not yet tested.
Decimal-binary conversion is not yet tested for accuracy.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 3, ... page 4

The program attempts to discriminate among:
  >FLAWs, like lack of a sticky bit,
  >SERIOUS DEFECTs, like lack of a guard digit, and
  >FAILUREs, like 2+2 = 5 .
Failures may confound subsequent diagnoses.

The diagnostic capabilities of this program go beyond an
earlier program called “Machar”, which can be found at the
end of the book “Software Manual for the Elementary Functions”
(1980) by W. J. Cody and W. Waite. Although both programs
try to discover the radix (b), precision (p) and
range (over/underflow thresholds) of the arithmetic, this
program tries to cope with a wider variety of pathologies
and to say how well the arithmetic is implemented.
The program is based upon a conventional radix
representation for floating-point numbers,
but also allows for logarithmic encoding (b = 1)
as used by certain early wang machines.


To continue diagnosis, press return.
Diagnosis resumes after milestone # 7, ... page 5

Program is now RUNNING tests on small integers:
FAILURE: violation of 240/3 = 80 or 240/4 = 60 or 240/5 = 48 .
```

```
Searching for radix and precision...
Radix = 2.
Closest relative separation found is 3.55271368E-15
Recalculating radix and precision
confirms closest relative separation .
Radix confirmed.
The number of significant digits of radix 2. is 48.00
Test for extra-precise subexpressions:
SERIOUS DEFECT: disagreements among the values X1, Y1, Z1
respectively 0.3552714E-14, 0.0000000E+00, 0.3552714E-14
are symptoms of inconsistencies introduced by extra-precise
evaluation of allegedly “optimized” arithmetic
subexpressions. Possibly some part of this
test is inconsistent; PLEASE NOTIFY KARPINSKI !
That feature is not tested further by this program.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 30, ... page 6

Subtraction appears to be normalized as it should.
Checking for guard digits in multiply divide and subtract.
DEFECT: division lacks a guard digit so error can exceed 1 ulp
or 1/3 and 3/9 and 9/27 may disagree.
SERIOUS DEFECT: subtraction lacks a guard digit so cancellation is obscured.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 40, ... page 7

Checking for rounding in multiply, divide and add/subtract:
Multiplication is neither chopped nor correctly rounded.
Division is neither chopped nor correctly rounded.
Add/subtract neither chopped nor correctly rounded.
Sticky bit used incorrectly or not at all.
FLAW: lack(s) of guard digits or failure(s) to correctly round or chop
(noted above) count as one flaw in the final tally below.

Does multiplication commute? Testing if x*y = y*x for 20 random pairs:
No failure found in 20 randomly chosen pairs.

Running tests of square root...
Testing if sqrt(x*x) = x for 20 integers x.
Found no discrepancies.
Sqrt has passed a test for monotonicity.
Testing whether sqrt is rounded or chopped:
Square root is neither chopped nor correctly rounded.
Observed errors run from -0.1000000E+01 to 0.5000000E+00 ulps.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 90, ... page 8

Testing powers z^i for small integers z and i :
Start with 0.**0 .
Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 90.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 90, ... page 9
```

```
Testing powers z^i for small integers z and i :
No discrepancies found.

Seeking underflow threshold and min positive number:


FAILURE: positive expressions can underflow to an allegedly
    negative value z0 that prints out as 0.00000000E+00
    but -z0, which should then be positive, isn’t; it prints out as
0.00000000E+00
Since comparison denies PHONY0 = 0,
 evaluating ( PHONY0 + PHONY0 ) / PHONY0 should be safe;
Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 115.
This is a VERY SERIOUS DEFECT.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 115, ... page 10

Smallest strictly positive number found is minpos = 0.00000000E+00
Since comparison denies MINPOS = 0,
 evaluating ( MINPOS + MINPOS ) / MINPOS should be safe;
Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 121.
This is a VERY SERIOUS DEFECT.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 121, ... page 11

Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 122.
The underflow threshold is 0.00000000E+00 , below which
calculation may suffer larger relative error than merely roundoff.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 130, ... page 12

since underflow occurs below the threshold =
     ( 2.00000000E+00)^( -8.19200000E+03) ,
only underflow should afflict the expression
     ( 2.00000000E+00)^( -1.63840000E+04) ;
actually calculating it yields
 0.00000000E+00
This computed value is O.K.
Testing x^((x+1)/(x-1)) vs. exp(2) = 0.73890561E+01 as x -> 1.
Accuracy seems adequate.
Testing powers z^q at four nearly extreme values:
No discrepancies found.


To continue diagnosis, press return.
Diagnosis resumes after milestone # 160, ... page 13
```

```
Searching for overflow threshold:
Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 161.
Can “ z = -y ” overflow? trying it on y = -1.36343517+2465
Seems O.K.
Overflow threshold is v = R
There is no saturation value because
the system traps on overflow.
No overflow should be signaled for v*1 =
                        R
            nor for v/1 =
                        R
Any overflow signal separating this * from one above is a DEFECT.

To continue diagnosis, press return.
Diagnosis resumes after milestone # 190, ... page 14

FLAW: unbalanced range; UFLTHR * V = 0.25000E+00 IS TOO FAR FROM 1.

What messages and/or values does division by zero produce?
About to compute 1/0...
Is this a program restart after failure (1)
or a start from scratch (0) ?
Restarting from milestone 211.
About to compute 0/0...
Trying to compute 0/0 produces 1.0000000E+00


To continue diagnosis, press return.
Diagnosis resumes after milestone # 220, ... page 15

The number of FAILUREs encountered = 2
The number of SERIOUS DEFECTs discovered = 4
The number of DEFECTs discovered = 1
The number of FLAWs discovered = 2
The arithmetic diagnosed has unacceptable Serious Defects.
Potentially fatal FAILURE may have spoiled this program’s subsequent diagnoses.
End of Test.
```

VAX Output from Run of Paranoia, Single Precision

```
Is this a program restart after failure (1)
or a start from scratch (0) ?
0
A Paranoid Program to Diagnose Floating-point Arithmetic
    ... Single-Precision Version ...
Lest this program stop prematurely, i.e. before displaying
 “End of Test”
 try to persuade the computer NOT to terminate execution
 whenever an error such as Over/Underflow or Division by
 Zero occurs, but rather to persevere with a surrogate value
 after, perhaps, displaying some warning. If persuasion
 avails naught, don’t despair but run this program anyway
 to see how many milestones it passes, and then run it
 again. It should pick up just beyond the error and
 continue. If it does not, it needs further debugging.

Users are invited to help debug and augment this program
so that it will cope with unanticipated and newly found
compilers and arithmetic pathologies.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 0, ... page 1


Please send suggestions and interesting results to
    Richard Karpinski
    Computer Center U-76
    University of California
    San Francisco, CA 94143-0704
    USA

In doing so, please include the following information:
    Precision: Single;
    Version: 31 July 1986;
    Computer:

    Compiler:

     Optimization level:

    Other relevant compiler options:


To continue diagnosis, press return.

Diagnosis resumes after milestone # 1, ... page 2
```

```
BASIC version (C) 1983 by Prof. W. M. Kahan.
Translated to FORTRAN by T. Quarles and G. Taylor.
Modified to ANSI 66/ANSI 77 compatible subset by
Daniel Feenberg and David Gay.
You may redistribute this program freely if you
acknowledge the source.


Running this program should reveal these characteristics:
b = radix ( 1, 2, 4, 8, 10, 16, 100, 256, or ... ) .
p = precision, the number of significant b-digits carried.
u2 = b/b^p = one ulp (unit in the last place) of 1.000xxx..
u1 = 1/b^p = one ulp of numbers a little less than 1.0.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 2, ... page 3

g1, g2, g3 tell whether adequate guard digits are carried;
1 = yes, 0 = no; g1 for mult., g2 for div., g3 for subt.
r1,r2,r3,r4 tell whether arithmetic is rounded or chopped;
0=chopped, 1=correctly rounded, -1=some other rounding;
r1 for mult., r2 for div., r3 for add/subt., r4 for sqrt.
s=1 when a sticky bit is used correctly in rounding; else s=0.
u0 = an underflow threshold.
e0 and z0 tell whether underflow is abrupt, gradual or fuzzy
v = an overflow threshold, roughly.
v0 tells, roughly, whether infinity is represented.
Comparisons are checked for consistency with subtraction
and for contamination by pseudo-zeros.
Sqrt is tested. so is y^x for (mostly) integers x .
Extra-precise subexpressions are revealed but not yet tested.
Decimal-binary conversion is not yet tested for accuracy.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 3, ... page 4

The program attempts to discriminate among:
  >FLAWs, like lack of a sticky bit,
  >SERIOUS DEFECTs, like lack of a guard digit, and
  >FAILUREs, like 2+2 = 5 .
Failures may confound subsequent diagnoses.

The diagnostic capabilities of this program go beyond an
earlier program called “Machar”, which can be found at the
end of the book “Software Manual for the Elementary Functions”
(1980) by W. J. Cody and W. Waite. Although both programs
try to discover the radix (b), precision (p) and
range (over/underflow thresholds) of the arithmetic, this
program tries to cope with a wider variety of pathologies
and to say how well the arithmetic is implemented.
The program is based upon a conventional radix
representation for floating-point numbers,
but also allows for logarithmic encoding (b = 1)
as used by certain early wang machines.
```

```
To continue diagnosis, press return.

Diagnosis resumes after milestone # 7, ... page 5

Program is now RUNNING tests on small integers:
-1, 0, 1/2 , 1, 2, 3, 4, 5, 9, 27, 32 & 240 are O.K.

Searching for radix and precision...
Radix = 2.
Closest relative separation found is 5.96046448E-08
Recalculating radix and precision
confirms closest relative separation .
Radix confirmed.
The number of significant digits of radix 2. is 24.00
Test for extra-precise subexpressions:
Subexpressions do not appear to be calculated
 with extra precision.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 30, ... page 6

Subtraction appears to be normalized as it should.
Checking for guard digits in multiply divide and subtract.
These operations appear to have guard digits as they should.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 40, ... page 7

Checking for rounding in multiply, divide and add/subtract:
Multiplication appears to be correctly rounded.
Division appears to be correctly rounded.
Add/subtract appears to be correctly rounded.
checking for sticky bit:
Sticky bit used incorrectly or not at all.

Does multiplication commute? Testing if x*y = y*x for 20 random pairs:
No failure found in 20 randomly chosen pairs.

Running tests of square root...
Testing if sqrt(x*x) = x for 20 integers x.
Found no discrepancies.
Sqrt has passed a test for monotonicity.
Testing whether sqrt is rounded or chopped:
Square root appears to be correctly rounded.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 90, ... page 8

    Testing powers z^i for small integers z and i :
    Start with 0.**0 .
    %MTH-F-UNDEXP, undefined exponentiation
user PC 00005B99
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name routine name line rel PC abs PC
```

```
0000C328 0000C328
 00033B23 00033B23
 PWRCMP    PWRCMP     32    00000019 00005B99
 POWER    POWER      57    00000087 000069C7
 SPREC$MAIN   SPREC$MAIN       257  000004AE 00004AAE
 $ run sprec
  Is this a program restart after failure (1)
  or a start from scratch (0) ?
 1
  Restarting from milestone 90.

  To continue diagnosis, press return.

  Diagnosis resumes after milestone # 90, ... page 9

  Testing powers z^i for small integers z and i :
  No discrepancies found.

  Seeking underflow threshold and min positive number:
  Smallest strictly positive number found is minpos = 2.93873588E-39
  Since comparison denies MINPOS = 0,
evaluating ( MINPOS + MINPOS ) / MINPOS should be safe;
what the machine gets for ( MINPOS + MINPOS ) / MINPOS is
 0.2000000E+01
 This is O.K. provided over/underflow has not just been signaled.
 FLAW: x = 0.40407618E-38 is unequal to z = 0.29387359E-38 ,
 yet x-z yields 0.0000000E+00
 Should this not signal underflow, this is a SERIOUS
 DEFECT that causes confusion when innocent statements like
 if (x.eq.z) then ... else ... ( f(x)-f(z) )/(x-z) ...
 encounter division by zero although actually x/z = 1 + 0.37500000E+00
 The underflow threshold is 0.29387359E-38 , below which
 calculation may suffer larger relative error than merely roundoff.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 130, ... page 10

since underflow occurs below the threshold =
     ( 2.00000000E+00)^( -1.28000000E+02) ,
only underflow should afflict the expression
     ( 2.00000000E+00)^( -2.56000000E+02) ;
actually calculating it yields
 0.00000000E+00
This computed value is O.K.
Testing x^((x+1)/(x-1)) vs. exp(2) = 0.73890557E+01 as x-> 1.
Accuracy seems adequate.
Testing powers z^q at four nearly extreme values:
No discrepancies found.


To continue diagnosis, press return.

Diagnosis resumes after milestone # 160, ... page 11
```

```
Searching for overflow threshold:
 %SYSTEM-F-FLTOVF_F, arithmetic fault, floating overflow at PC=00005EF2,
PSL=03C0002A
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name     line    rel PC     abs PC

OVERF    OVERF           104    0000026A 00005EF2
SPREC$MAIN    SPREC$MAIN            276  000004CF 00004ACF
$ run sprec
 Is this a program restart after failure (1)
 or a start from scratch (0) ?
1
 Restarting from milestone 161.
 Can “ z = -y ” overflow? trying it on y = -8.50705917E+37
 Seems O.K.
 Overflow threshold is v = 1.70141173E+38
 There is no saturation value because
 the system traps on overflow.
 No overflow should be signaled for v*1 =
1.70141173E+38
 nor for v/1 =
1.70141173E+38
Any overflow signal separating this * from one above is a DEFECT.

To continue diagnosis, press return.

Diagnosis resumes after milestone # 190, ... page 12


What messages and/or values does division by zero produce?
About to compute 1/0...
%SYSTEM-F-FLTDIV_F, arithmetic fault, floating divide by zero at PC=00009EDC,
PSL=03C
00022
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name     line    rel PC     abs PC

ZEROS    ZEROS           52    00000074 00009EDC
SPREC$MAIN  SPREC$MAIN             282  000004E0 00004AE0
$ run sprec
 Is this a program restart after failure (1)
 or a start from scratch (0) ?
1
 Restarting from milestone 211.
About to compute 0/0...
%SYSTEM-F-FLTDIV_F, arithmetic fault, floating divide by zero at PC=00009F37,
PSL=03C
00022
%TRACE-F-TRACEBACK, symbolic stack dump follows
module name     routine name     line    rel PC     abs PC

ZEROS    ZEROS           59    000000CF 00009F37
SPREC$MAIN  SPREC$MAIN             282  000004E0 00004AE0
$ run sprec
Is this a program restart after failure (1)
or a start from scratch (0) ?
1
Restarting from milestone 212.
```

```
To continue diagnosis, press return.

Diagnosis resumes after milestone # 220, ... page 13

The number of FLAWs discovered = 1
The arithmetic diagnosed seems Satisfactory though flawed.
End of Test.
FORTRAN STOP
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE February 1995	3. REPORT TYPE AND DATES COVERED Technical Memorandum

4. TITLE AND SUBTITLE Transferring Ecosystem Simulation Codes to Supercomputers	5. FUNDING NUMBERS 233-01-03-05
6. AUTHOR(S) J.W. Skiles* and C.H. Schulbach	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Ames Research Center Moffett Field, CA 94035-1000	8. PERFORMING ORGANIZATION REPORT NUMBER A-94142
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Aeronautics and Space Administration Washington, DC 20546-0001	10. SPONSORING/MONITORING AGENCY REPORT NUMBER NASA TM-4662

11. SUPPLEMENTARY NOTES
Point of Contact: J.W. Skiles, c/o Ames Research Center, MS 239-20, Moffett Field, CA 94035-1000; (415) 604-3614
*Johnson Controls World Services, Inc., Cape Canaveral, Florida.

12a. DISTRIBUTION/AVAILABILITY STATEMENT Unclassified — Unlimited Subject Category 61	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Many ecosystem simulation computer codes have been developed in the last twenty-five years. This development took place initially on main-frame computers, then mini-computers, and more recently, on micro-computers and workstations. Supercomputing platforms (both parallel and distributed systems) have been largely unused, however, because of the perceived difficulty in accessing and using the machines. Also, significant differences in the system architectures of sequential, scalar computers and parallel and/or vector supercomputers must be considered. We have transferred a grassland simulation model (developed on a VAX) to a Cray Y-MP/C90. We describe porting the model to the Cray and the changes we made to exploit the parallelism in the application and improve code execution. The Cray executed the model 30 times faster than the VAX and 10 times faster than a Unix workstation. We achieved an additional speedup of 30 percent by using the compiler's vectorizing and "in-line" capabilities. The code runs at only about 5 percent of the Cray's peak speed because it ineffectively uses the vector and parallel processing capabilities of the Cray. We expect that by restructuring the code, it could execute an additional six to ten times faster.

14. SUBJECT TERMS Ecosystem modeling, Porting to supercomputers, Supercomputers	15. NUMBER OF PAGES 44
	16. PRICE CODE A03

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT